Development of an Augmented Reality Testing Platform for Collaborative Robots

Undergraduate Honors Thesis

Presented in Partial Fulfillment of the Requirements for Graduation with Distinction in the Department of Mechanical Engineering at The Ohio State University

by Cameron Spicer

May 2018

Advisor: Haijun Su, Ph.D.

Abstract

In manufacturing, the reliance on robotics is increasing. Current industrial robots are designed for large production lines but suffer from issues, such as safety concerns and complicated programming. Collaborative robots or “co-robots” may solve these problems to make robotics useful for small and medium scale operation. These co-robots accomplish this by working with a human operator. The operator performs precise tasks, and the co-robot preforms repetitive tasks.

The purpose of this study was to develop a testing platform for co-robots using the Microsoft Hololens and augmented reality (AR) device. AR is an emerging technology, which creates virtual holograms a user can interact with in the real world. The testing platform uses AR to simulate a manufacturing environment and an interactive co-robot, which are modeled in Unity, a video game platform. Unity’s built-in physics engine was leveraged to create hinge joints, which emulate the co-robot’s motion. A test environment was chosen that simulated the assembly of an automobile front knuckle after consulting Honda Manufacturing. The user directs the robot on where to pick up and place cumbersome parts to assemble the knuckle. The parts that make up the knuckle are heavy for humans, so an assisted lifting device is typically used. The robot would replace the device removing the human guidance and increasing production.

This research shows the Hololens can be used to effectively model a manufacturing environment with an interactive co-robot. Additionally, a robust platform for research and programming of a co-robot can be created in AR. This research was the first step in developing a platform for testing co-robots in any manufacturing environment using AR. With more development, this platform would allow for companies to test co-robots and hopefully, increase the use of robotics in small and medium scale manufacturing environments.

Acknowledgements

I would like to thank my research advisor Dr. Haijun Su for all the support he has given me over this year. He helped push the project along when it was stuck and used his resources for my benefit. Without his ideas, this project would have never been started.

Additionally, I am grateful for Anil Turkkan and Tyler Morrison. These two grad students helped me with multiple problems. They encouraged the project along and spent their time working and supporting me.

I would like to express my appreciation for Roger Kassouf who is my research partner. He has collaborated with me throughout this entire project and will be continuing this project in the future. He was always there to bounce ideas off of and keep us on schedule. He has been great to have around and work with.

Robert Siston has mentored me from the start. He encouraged me to pursue undergraduate research and gave me advice on how to continue forward. He has helped me practice my presentation skills and encouraged me throughout this entire process.

Honda North Americas Inc. graciously gave a tour to the group of researchers in January of their manufacturing environment. Thanks to them for allowing us to see current uses for robotics and apply it to the simulations. The tour helped make the simulation more realistic and applicable to manufacturing. Specifically, thanks to José Carlos N Banaag for setting up the tour with Dr. Su.

I have a debt of gratitude to Jing Huang and Hanyuan Xu for starting this project over the summer. Their work helped greatly getting past the beginning bugs and intricacies with the Hololens.

Finally, I would like to recognize my two good friends, Adam Stevens and Taylor Burrowes. These two have assisted me through the entire process. They have watched my presentation more times than I can count and always gave good advice on how to improve. Moreover, they have taken time to proof read through the paper more than once.

Table of Contents

List of Figures

Chapter 1: Introduction

Robotics have been around for over 50 years. In this time, people have become more accustomed to seeing and working around robots. Robots started as research tools, but as the price of computers and sensors decreased, they became more powerful and robust. New unimaginable robots came to be used in a variety of fields. Robotics are seen in industry from autonomous cars, exploring space, and the medical field. These robots have been designed to take the jobs no one would want such as dirty, dangerous or monotonous jobs [1].

In 2016, there are 66 industrial robots per 10,000 employees in worldwide companies [2]. This is due to the increase in cost of a U.S. worker and an increasing emphasis on automation. This emphasis has increased the adaptability, precision and reliability of assemblies. The aim of increasing robots in manufacturing is to keep jobs in the U.S., make companies more competitive, and provide skilled jobs working with and programming robots. This allows for human robot teams, thus improving working conditions and safety.

Current manufacturing robots are most commonly thought of in large scale assembly lines, such as a car manufacturing plant. The most prevalent robot called a “manipulator” is a set of linked joints creating an arm that can manipulate objects. These robots are built for speed and precision, meaning they have a lot of force that could be delivered if they encounter an obstruction. For safety reasons, they are placed in a protective cage and must be shut down if maintenance is to be performed. Additionally, programming current industrial robots requires a skilled technician and days to reprogram. For small and medium scale production with small lot sizes, this can be costly and inefficient [2]. The way these robots become cost efficient is with large lot sizes and no need to reprogram the robots.

Figure 1: Car Assembly Line [3]

Robots are relatively simple and as such have a few limitations. The first limitation is a gripper design. The current grippers are not dexterous enough for picking up everyday items, such as a vacuum. This means it is hard for a robot to have multiple functions involving many different objects with various shapes. Another limitation of current robotics is the controls. The control system for robots must be thorough and designed for changes to the environment. However, many of them are not fully robust. Control systems are tested in warehouses moving products at 30 mph and in fully autonomous systems, like unmanned aerial vehicles. Finally, the robots should be flexible and able to complete multiple jobs. These robots should be able to work with humans to complete a wide variety of tasks.

To advance robotics flexibility, a new type of robot is being designed. This type of robot is called a collaborative robot (co-robot). These industrial machines are designed to work in direct cooperation with humans within a defined collaborative workspace [4]. The co-robot takes on the repetitive tasks; meanwhile, the operator can take on the detailed tasks that would be complicated to program a robot to complete.

Co-robots have the user interaction built into the design. The co-robot is first designed to be operated safely without a cage and close to the operator. A simple safety feature designed in these robots is the absence of pinch points and sharp edges. Additionally, sensors are added to the joints to measure the force on the robot. The force limiting devices stop the robot from injuring an operator or destroying the work area. In addition, the coding of the co-robots is designed to be easier and faster so it can adapt to any task. Research is allowing the robot to learn from human input, therefore, allowing anyone to give the robot instruction. The aim is to reduce the programming time of industrial robots from days to a single eight-hour shift [1].

There are a few co-robots on the market as well as a few are being used in research. The first of these is Baxter, as illustrated in Figure 2. Baxter is a two-arm robot with 7 degrees of freedom per arm. Baxter includes 5 cameras, sonar, and force feedback in the joints. Another collaborative feature is the eyes. These eyes move to tell the humans where the robot moves. Another type of co-robot on the market is the Universal Robots UR-10 seen in figure 4. This is closer to the standard industrial robot with 6 joints and a 51” reach. It is built for working close to humans and adapting to the factory floor following the ISO 10218 standard.

Figure 2: Baxter Commercially Available Co-robot [5]

Another emerging technology used in this research is virtual reality (VR) and augmented reality (AR). Virtual reality has been around for over 60 years which begun with the introduction of the sensoroma. This attempted to merge the technology of the time to turn a movie into a 3D production. In 2010, the Oculus Rift was created and started the current revolution in technology. Virtual reality is where a device creates a new world for users to explore. It does this with screens in front of the user's eyes with sensors to determine head movement [6]. A few years later in 2016, an AR device was created, called the Microsoft Hololens. Augmented reality takes virtual reality one step further. Holograms are created and placed in the real world, layering a virtual reality on top of the physical reality. The Hololens has a sensor suite of cameras, which can pick up gestures made by the user and the room geometries. These cameras allow for seamless interaction between the holograms and the environment.

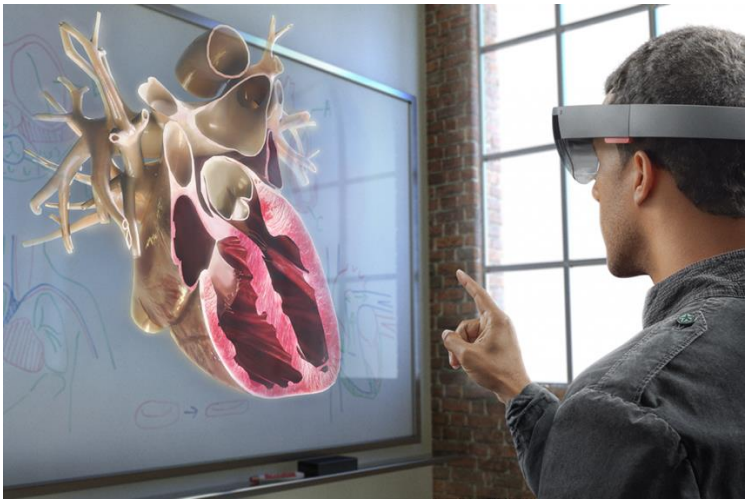

Figure 3: Hologram of the Human Heart with the Hololens [7]

Since the Hololens has only been around for two years, some companies are exploring its use in their markets. The first market testing AR was the video game industry. They are trying to design a system that can fully immerse people in the game to give them a unique experience. NASA's Jet Propulsion Laboratory has been using the device to take pictures from the Mars rover and lay them out for a human observer. The observer can look at the pictures more closely and circle things that stand out for further research. Additionally, NASA is using the Hololens to create and examine the designs for the next rover. In the architecture industry, apps have been created allowing designers to place buildings and see what a city block would look like with the buildings before they are built. The Hololens is being used as a training device for ThyssenKrupp to teach the employs on how to service their elevators. AR is a valuable tool for marketing at Volvo where customers can be shown the car as well as how the safety features work, which could not be conveyed easily in the standard brochure. The device can be used in the medical and education industries. Surgeons are using AR to project a real-time x-ray like image onto a patient. This allows the surgeon to see where all the organs are when performing a minimally invasive surgery. Also, to educate medical students, a model of the human body has been created to replace some cadaver training. The students can see the layers of organs as well as click on the organ to find out more [8]. The Hololens has been shown to be a powerful tool in a variety of industries and can become a powerful tool in manufacturing.

1.1 Focus of Thesis

Research into the use of collaborative robots in a manufacturing setting is limited by the time and availability of the equipment. The focus of the thesis is to develop an augmented reality environment that allows for collaborative robots to be tested in manufacturing settings.

1.2 Significance of Research

Manufacturing is an important part of society in America. It accounts for 12% of the United States GDP. However, with American worker numbers in manufacturing decreasing, robots are being tasked to take over the load. Therefore, this $9 billion industry is growing at 9% each year [1]. Most of this growth is due to large scale manufacturing, which is suited for current industrial robots. Collaborative robots are being researched to take up the roles in medium and small-scale manufacturing the current robots cannot do.

The development of a test platform for collaborative robotics is essential to further research. The platform will open doors for researchers because more labs can test robotics with the only cost being the HoloLens. Also, labs will be able to test various scenarios quickly and efficiently. Unity will allow for levels to be created, in which the user can speedily choose a different scenario. The platform will allow for the creation of different scenarios in a plug and play fashion. Augmented reality will add a more realistic feel to the scenarios as compared to virtual reality and the same scene shown on a computer screen.

This research will help industry in many ways. First, the platform can be configured to validate programming in a manufacturing environment. The augmented reality device could place holograms over the manufacturing floor to run tests of the programming before the robots are ran in real life. Additionally, the platform could be used to train operators with the collaborative robot they will be working with on the floor reducing production downtime. These tests are required to have a feasible robot for small and medium scale production.

AR research, like this, has an overall industry goal to increase the adoption of collaborative robotics in the medium and small-scale manufacturing environments. With more research into collaborative robotics, the United States can lead the world in manufacturing and robotics.

1.3 Overview of Thesis

The thesis has five chapters. In Chapter 2, the process of programming the robot will be detailed. The method of using the built-in Unity functions to make the physics engine work will be explained. In Chapter 3, the environment setup and programming will be discussed. The use of the Unity game engine and the process of defining the environment will also be explicated. The testing and evaluation of the environment will be deliberated in Chapter 4. Finally, the last chapter is the conclusion, where the results and future plans will be compiled.

Chapter 2: Development of The Robot

The collaborative robot being used for this research is the Universal Robotics UR-10. This is a six degree of freedom (DOF) robotic arm. The arm has a max payload of 10 kg with a max working radius of 1.3 meters. This robot can be seen in Figure 4. It is designed to be a universal solution for manufacturers, but currently only performs a few main tasks. The first is machine tending. These robots are placed close to a CNC machine and shift the parts in and out of the machine to continue production. Additionally, these robots are being fitted with cameras to perform a quality inspection of parts. They can perform pick-and-place tasks, such as moving an item from one assembly line to another. Finally, they can be used for assembly of items handling plastics, metals and wood materials. Since this robot is very versatile, it was chosen to be used in the simulation and the CAD files can be found online.

The robot is constructed from seven parts. The first part is the base. This part is connected to the ground and secures the robot from moving. Attached to this part is the shoulder. The shoulder connects the longest arm, called the upper arm, to the base. Then, connected to the upper arm is the lower arm. Finally, the last three parts give the arm multiple degrees of motion. There is the wrist 1 part that is connected to the lower arm and the wrist 2 part connected to wrist 1. The final piece is the tool flange, also known as the end effector. This is the part that would connect to grabbers or other assorted parts.

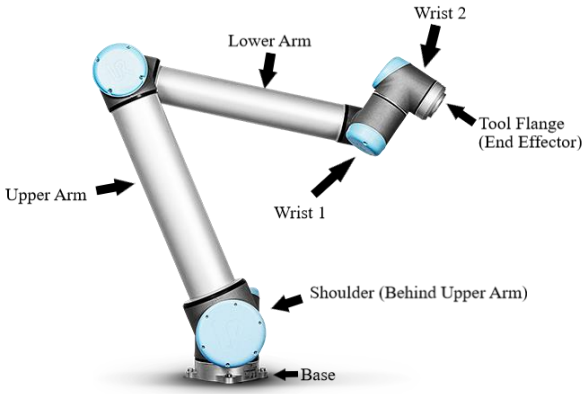

Figure 4: UR-10 Robot [9]

Over the summer, two students started the research. They had previously worked to set up the Hololens and work with Unity. This saved a lot of initial time of learning and using Unity with the Hololens. A robot model had been constructed to work, but it had a few errors such as the look and the use of the dragging feature. A simple demo had been created to pick and place car parts to assemble a car. This was unrealistic because the car parts would be heavier than the robot could handle and not a realistic manufacturing environment.

2.1 Construction of the Robot

To create a working robot in Unity that can accept a user input, a few steps were followed. The first step is to import the parts into the workspace. These parts were placed in the robot's starting position as seen in Figure 5. The next step was to determine how the robot would move. There are two different ways of telling the parts of the robot where to go. The first way is creating a code that utilizes inverse kinematics. This code is math heavy and has a few instances where the solution is undefined or multiple solutions are present. For this reason, the second method was chosen for the robot. This method uses a dynamic simulation that is built into Unity. This is called the physics engine. This engine takes the joint location and uses this to solve where the robot parts should be in space when moving the end effector of the robot.

To use the physics engine, each part must be set to a Rigidbody. This allows the physics engine to calculate the movement of the parts based on the mass, drag and other properties. One key point is the Is_Kinematic property, which is a toggled check box under the Rigidbody menu. When checked, this stops forces and dragging via gestures from acting on the part. Next, Unity must be told the type of joint and the placement of the joints in the robot. There are a few different joint types to choose from; the spring joint, fixed joint and hinge joint were explored in this research. The spring joint connects two parts by a spring. The fixed joint is used when two parts are connected and not supposed to rotate or translate relative to each other. Finally, the hinge joint connects two parts with a revolute joint.

Hinge joints were added to each of the six areas of movement in the arm. To create the joint, the component is added to one of the parts involved in the joint. Then, the second part is filled in as a connected body. Unity will then autofill the anchor, axis and connected anchor of the joint. The anchor placement defines where the joint moves about, while the axis defines what it revolves around. Unity sets the axis, but it must be checked to ensure the correct rotation. There are other settings that can modify these joints like adding a spring force or determining if the joint breaks. One of these settings used in the simulation is the damper setting. This setting was set to infinity to damp out movements that would not be present in the true robot.

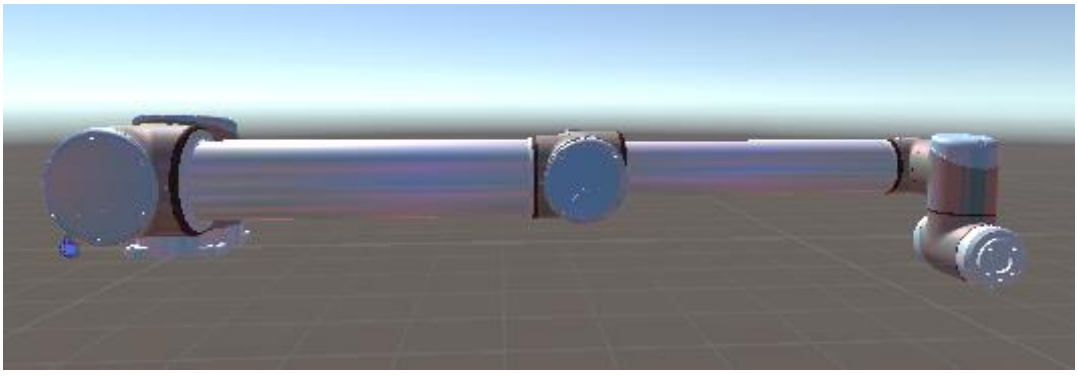

Figure 5: UR-10 in the Starting Position

On the first design iteration of the robot, 11 parts made up the robot corresponding to the CAD files from Universal Robotics. This required the use of fixed joints to create the Upper and Lower Arm. These joints were eliminated in the second version of the robot. This was due to their tendency to break easily. The assemblies of the upper and lower arm, in SOLIDWORKS, was saved as a part file. This merged the three components needed into one and eliminating fixed joints. The new version of the robot eliminated redundancy and the joints, which caused the most trouble. Now in this iteration, there are only seven parts.

The next step to utilize the physics engine is to set colliders. These are components that tell the engine where the part is so that it will not intersect other parts. For this to work, each part must have a collider attached to them. There are a few collider types. The box collider is the least computationally heavy collider, so it was chosen for this project. This places a box around the part to determine collisions. The box collider is the least accurate, but the trade-off is speed, making it ideal for this simulation. A box collider was added to each component of the robot.

The subsequent step is to make the robot look realistic. This was done by changing the colors of the robot. The students over the summer were able to make a grey robot. The colors looked correct in SOLIDWORKS, but when exported to Unity through 3DSmax the colors were stripped. This was fixed by changing the export process. The export process to Unity is explained in Appendix B. This process created a more realistic hologram of the robot.

The final item needed for a moveable robot arm hologram was the ability to control the arm. There are a few ways to accomplish this. One was to have the user click on buttons to move the robot in specified directions. Another is to allow the user to drag a waypoint for the robot to follow using motion planning. The method chosen for this iteration of the simulation was allowing the user to drag the robot itself through the motion. This was chosen for simplicity, and more features can be added in the future. The dragging was accomplished by allowing the user of the Hololens to use the click gesture on the tool flange and drag it around the working area. Luckily there is a C# script called “Hand Draggable” in the Mixed Reality Tool Kit. To make this script work, the InputManager game object must be added to the scene. This object is invisible and contains a few scripts needed to update the scene. Within the Hand Draggable script, the position lerp speed must be set to 0.05 and the rotation lerp speed to 0.1. These settings slow the max speed the robot can be dragged. This reduces the chance of the human user dragging the robot past the physical working area. Once these are set, the application can be compiled and run. The user clicks on the box collider of the tool flange, and then can drag the robot until they unclick or leave the viewing area of the Hololens.

2.2 Fixing the Robot

Over the summer, a rudimentary dragging system was created and was working. But when a user dragged the robot quickly or out of the robot working area an unwanted side effect would occur where the robot would drag apart and break at the hinge joints. In the first version of the robot, the fixed joints would break apart quickly, but this was resolved by eliminating the fixed joints. Once the fixed joints were eliminated in version two, the hinge joints would break apart and the parts of the robot would shake uncontrollably. This effect can be seen in Figure 6.

Figure 6: Broken Arm after Dragging to Quickly

The shaking uncontrollably was a large problem for the simulation. This unrealistic effect forces the user to restart the program to continue testing. There were two causes of this problem, the speed of dragging and dragging outside of the working area. To fix the first part of the problem, the speed of dragging from the first iteration of the simulation was decreased by decreasing the lerp speed. To fix the second reason for this unwanted effect, dragging must be stopped when the end effector approaches the end of the working area. Since the UR-10 is a single arm robot, the working space is a sphere radius 1.3 meters in length. This was accomplished by modifying the script responsible for dragging the end effector, called HandDraggable. The first solution was to make a code that calculates the length the tool flange from the base to see if it exceeds the 1.3 meters.

To modify the code, a function was created that was called each time before the update of the scene. The length is calculated using Vector.Distance as seen in the code snippet below.

```
float dist = Vector3.Distance(GameObject.Find("Tool Flange").transform.position,
GameObject.Find("Shoulder").transform.position);
```

This finds the two GameObjects and calculates the true point-to-point distance between the origin of each part. In the code, on each update the position is stored before moving. Then, when a point exceeds the distance, the tool flange is moved back to the original position and the StopDragging() function is called. This resets the tool flange so the user can drag the robot again without breaking the robot. This code can be seen in Appendix E. This code was made to fit most robot arms when the working volume is a sphere, but robots that are not manipulators may have a different working volume.

Another way to approach the problem is to check the forces the joints are experiencing. However, Unity does not offer a way to obtain the forces experienced in the joints. If the force could be found, then every robot brought into the simulator could be treated the same. Once a threshold was reached, the code would stop the dragging event and reset the robot. An attempt was made to utilize this method, but was unsuccessful. To get around Unity, a spring joint was used. A GameObject was attached to the robot's end effector through a spring joint. The spring joint allowed for a spring of a variable stiffness to pull the robot to the correct place. Then, the same code used above could check the distance between the new GameObject and the tool flange. By using Hooke's law, the force between the GameObject and the tool flange was obtained. This process was tested in the simulation, but if the dragging occurred to fast the joint still broke. Therefore, the boundary sphere method was used for this thesis.

Another aesthetic issue was when the robot was dragged and then stopped, the residual momentum caused the upper and lower arm to oscillate up and down, which was an unrealistic effect in the simulation. To eliminate this, two solutions were tried. The first was to increase the damping coefficient to the max value on the joints. This reduced the amplitude of the oscillations, but did not remove it. The second, better solution was to modify The Hand Draggable script to flip the Is_Kinematic toggle on each part of the robot when the user was not dragging. This enables the kinematic feature and eliminates all effects of forces on the parts when the robot is not being interacted with. This small change in the code removed the effect.

In this chapter, the designed robot hologram was discussed. This design iteration saw multiple upgrades. The first of these was eliminating the redundant joints to six hinge joints. This paired with the modification of the code reduced the robot's tendency to move uncontrollably and unrealistically. In addition, the colors of the robot were updated to make the robot appear more realistic. The final addition was to eliminate the oscillations in the upper and lower arm. With all these new upgrades, a better version of the robot was made for the simulation.

Chapter 3: Development of the Environment

For this thesis, a manufacturing environment similar to a real environment was desired. To explore typical manufacturing scenarios, a tour of the manufacturing environment at Honda North Americas Inc. was taken in January 2018. Honda manufactured the 2017 Civic in Marysville, Ohio. This assembly line was constructed by two moving lanes shaped like horse shoes. In some portions of the line, operators had to hop in the cars to mount different objects to the vehicle. Every station was dependent on each operator and robot finishing their task on time and not shutting down the line. Down time, even a few seconds, was costly to the company and the assembly line. If the line falls behind on production, employees had a working Saturday or Sunday. The front knuckle assembly was a special station not connected to the main moving lines.

The front car knuckle was a component located on the front axle. This part was responsible for taking the load of the car and allow for turning of the front tires. These tires were connected through a wheel hub to the knuckle and the suspension system. These parts were built to last and therefore, are heavy parts. The front drive train of a car can be seen in in Figure 7 with the front knuckle assembly surrounded by a box [10].

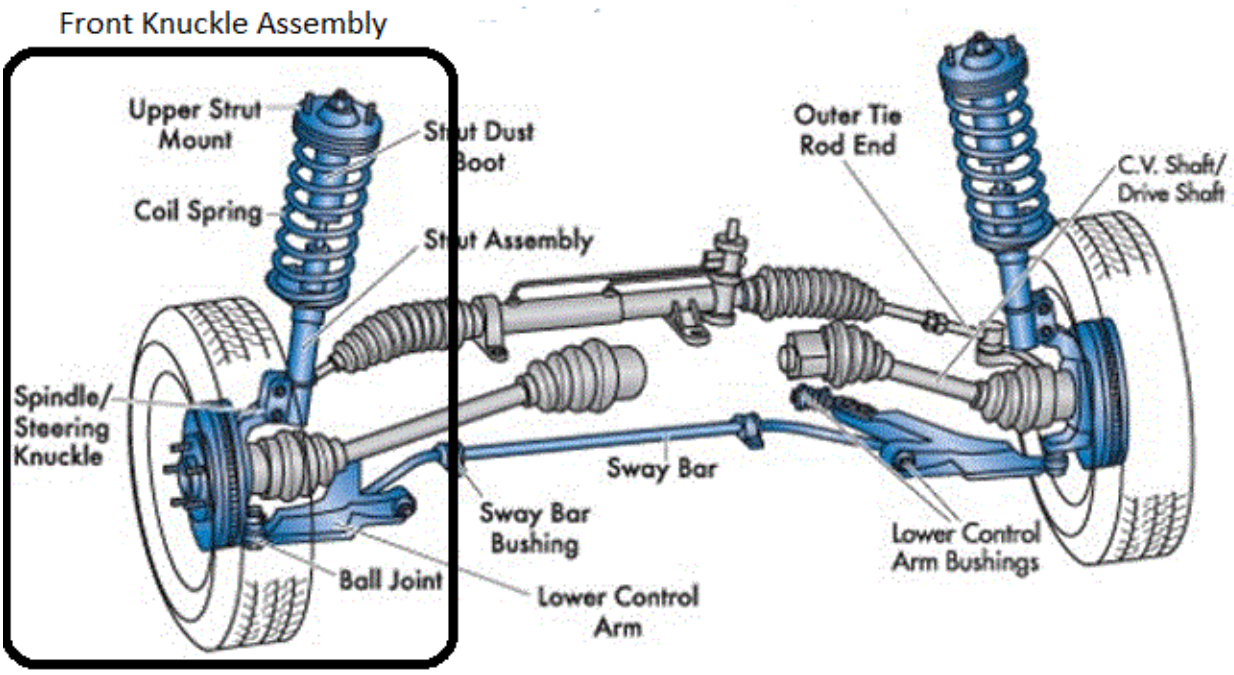

Figure 7: Typical Front Wheel Drive of a Car [11]

Because of the placement and tasks of the front knuckle assembly, this station was identified as a potential station that could benefit from human and robot collaboration. This would be a great proving ground for a robot, since the station could be modified without stopping the main line. If the robot were to break down, the full line would not be forced to stop. Then the human operators could take over the task while the robot is being fixed.

3.1 The Manufacturing Environment

The current knuckle assembly station was composed of two operators using assisted lifting devices, which hang from the ceiling. There were two assembly benches, each with a jig fixture that helps with assembly. Each operator had one fixture and one task associated with their fixture. Then behind the operator, there was a pallet holding the shocks and another pallet that holds the knuckle pieces. The layout can be seen in Figure 8.

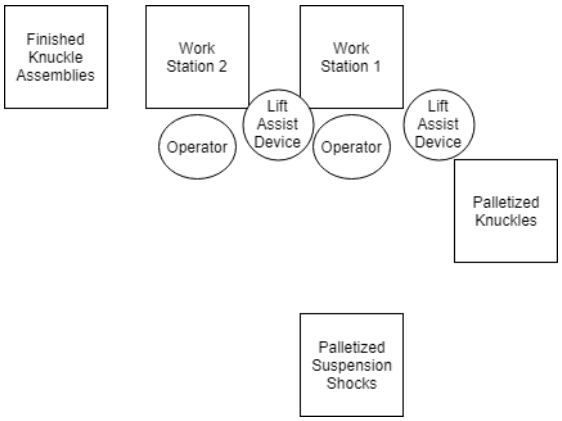

Figure 8: Original Factory Layout of Front Knuckle Station

The first step of the work flow was Operator One grabbing an assisted lifting device and placing the knuckle assembly in their jig. Then the hub and bearing assembly was bolted on and moved over to Station Two with the lift assist device. At Station Two, the other lift assisted device was used to grab the shock and place it in the jig. This was bolted on to the knuckle and the control arm is mounted to this assembly. The final assembly was moved to a pallet to be used in the main assembly line.

This task could be improved by adding a co-robot to reduce the time of assembly. The operator would take the task of bolting each piece together while the robot can take the place of the assisted lifting device. The robot would be able to grab items while the user focused on aligning and bolting the items together, reducing the time the operator must take to move components to the fixtures.

To add the UR-10 to this assembly process, the layout must be changed. The new layout can be seen in Figure 9. The original layout had sliding assisted lifting devices, whereas the robot would be mounted to a fixed point. Looking at the max working radius of the UR-10 of 1.3 meters, the best placement was right between the shocks and the work station. The operator was flipped to the other side of the table since the robot now needs to be where the operator used to be. One other advantage of this is the line of sight the operator has with the robot. The line of sight will help keep the operator at ease seeing the robot move versus the robot moving behind the back of the operator.

This new layout removed the second operator. This removal simplified the simulation to one robot and operator pair with the aim to investigate the interactions between the two. This eliminated the need for acquiring another Hololens and learning how to have multi-user simulations.

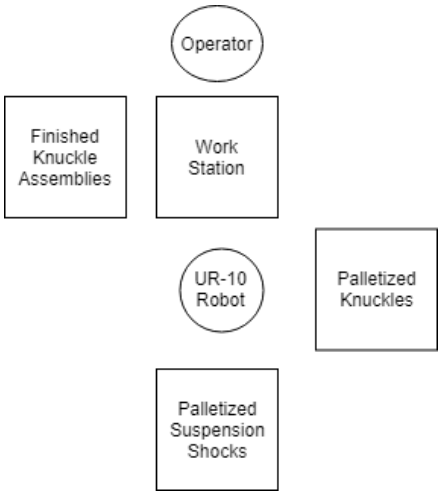

Figure 9: Changed Factory Layout Used in Simulation

The new task flow follows the same order as above, but with a robot performing the tasks in place of the assisted lifting device. The operator would not have to manually grab each shock and knuckle. The robot would be programmed to grab each item and deliver to the fixtures when the time is right. This configuration should reduce the process time by collaboration. The collaboration would be better ergonomically for the operator who does not have to do any heavy lifting or moving of the assisted lifting devices.

3.2 Setting Up the Simulation

To create the environment, the parts had to be made or sourced from a free CAD website, GrabCAD [12]. A standard pallet, knuckle and a shock were sourced from GrabCAD and combined with SOLIDWORKS parts to make up a shock pallet, a knuckle pallet and a work area with a jig. The jig was made in SOLIDWORKS in assembly with the hub and shock. Since the exact dimensions of the jigs used at Honda were unknown, this part was made to stand in their place. The two pallets and the assembly table were created to mimic the real designs used by Honda. Then these were imported into Unity and set up to fit the layout defined earlier.

The shock, in Figure 10, was 18 in long and had a max diameter of 3.5 in. These were cumbersome for a human to lift and that is why the assisted lifting device was used and why the robot would be delivering them to the operator. The knuckle assembly, in Figure 11, was composed of the knuckle and the hub. These were pre-assembled before this station as they were clumsily shaped and heavy for a human to carry.

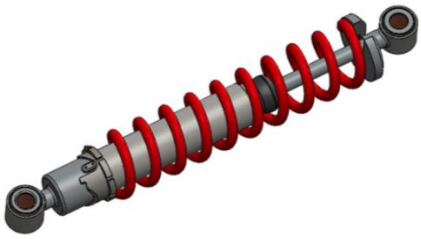

Figure 10: Shock

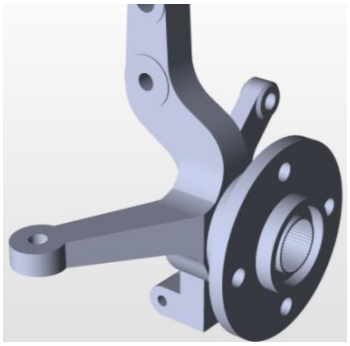

Figure 11: Knuckle Assembly

All the items were imported into Unity scene. When the scene was run the first time, it was obvious that the Hololens lagged in the visual renderings. This was investigated using the Windows Device Portal, showing the frame rate drops instantly from 60 frames per seconds to 30 frames per second when any hologram enters the view. This was typical for the Hololens. When this scene was loaded, the Hololens was running at 20 frames per second. This was due to the shocks and knuckles complex geometry. These parts initially were made with medium render quality, but this caused the decrease in frame rate. This was fixed by decreasing the render quality output of 3DSMax and the import settings in Unity modified. The mesh compression setting was changed from off to high. This fixed the issue of lagging and frame rate decreases in the scene.

Now that all the items needed for the assembly task had been imported into Unity, the simulation could be set up and coded. Unity's inspector was used to move the items into their starting positions in the simulations and can be seen in Figure 12. Additionally, each part had a box collider attached to them. For the parts not expected to move during the simulation, only a box collider must be attached to stop items from moving through them. Unity optimized the computation when this was done because these items were stationary. For the moving items, a Rigidbody component was added to the parts. The Rigidbody allowed for forces to act on the shocks and wheel knuckle. Unity used this data to attempt to optimize the computation time since these Rigidbodies were expected to move.

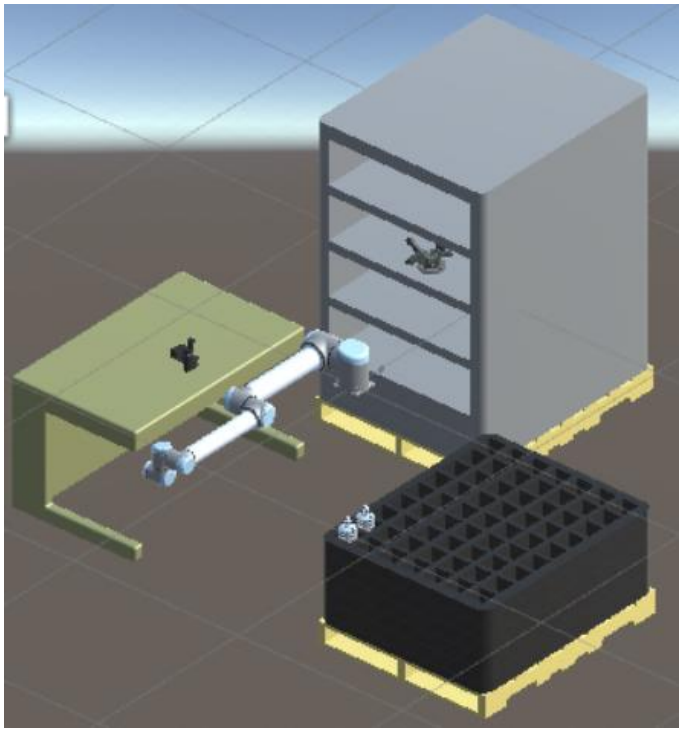

Figure 12: Environment Setup in Unity Scene

When Unity loads normally, the origin of the scene was placed at some arbitrary point ahead of the user and was not consistent between different Unity loads. To move the environment and reset the origin, the spatial perception feature of Hololens was used. This feature used the Hololens built-in cameras to map out the room. A mesh visualization could be seen in Figure 13. Then, the environment holograms could be moved and placed on the floor. Dragging the holograms moves the origin to match the desired real-world origin. Figure 14 shows a box placed on a table using the spatial perception.

Figure 13: Mesh Created from Spatial Perception

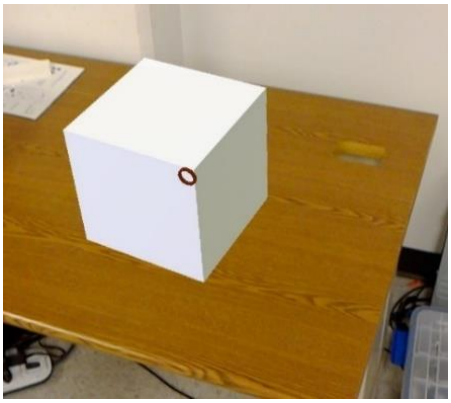

Figure 14: Block Placed on a Table Using Spatial Perception

The code used in this thesis is a modified code from the Virtual Reality Tool Kit. This script is called Tap to Place which requires a second script called Interpolator. In the Input Manager script, the empty GameObject, the World Anchor Manager, Spatial Mapping Manager, and Spatial Mapping Observer scripts must be added as components for the spatial perception to work. These three scripts monitor the states of variables in the scene, while the simulator was running.

The Tap to Place script waited for the user to click the object and then entered the placing mode. During this time, the cursor and object were locked at some distance from the user and was moved by the user gazing in differing directions. The Hololens cameras were continually scanning the environment during this phase, which can be costly for the processor and GPU. The code was configured to only run these processes when in the placing mode. This script found the parent of the object had been clicked and fed this information to the Interpolator script. The Tap to Place script, found in Appendix F, then looked for an air tap to exit the placing mode.

The Interpolator script received the parent information. Then, as the user moved their head, this script would interpolate all the other objects in the parent. This seems to work well when pieces were in the environment and relationships did not have to be preserved precisely. This script did not work well when joints were involved, such as the robot. When the robot was moved around, the Is-Kinematic feature was enabled. This stopped the robot from moving when being dragged around the environment. The issue came when Is-Kinematic was turned to false, the robot parts would shake uncontrollably until the robot was not usable. This was due to the interpolator moving the parts close to the correct position, but not precisely. To fix this, the Interpolator script was modified. The parent information received was changed into a public variable, which could be modified in Unity's inspector window. This would hard code the parent as the part that was moving. This maintained the relationship between the parts of the robot because the interpolator would not have to interpolate the parts, just move the parent. This hard code was only needed when joints and precise placement was needed. This modified script can be found in Appendix G. The scripts were applied to the robot model, which the robot on a real table can be seen in Figure 15.

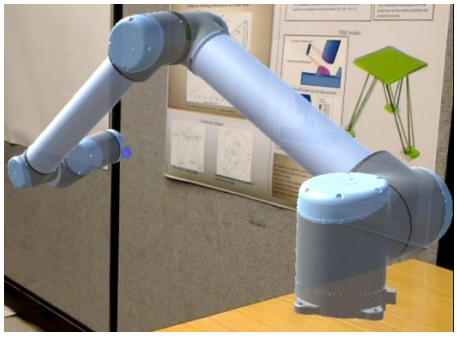

Figure 15: Robot Placed on a Table

3.3 Robot Interactions with the Environment

Now that the robot and the environment had been set up, the robot must be coded to interact with the environment. Since all parts had colliders, the robot should not be able to move through any part. This was the first step in making the interactions seem realistic in the simulation. However, the robot must be able to pick and place objects in the simulation. This action would occur when the end effector of the robot was close enough to the part to pick it up. Then the robot released the part when the part was close enough to drop off. The distance needed to be deemed close enough for the action and was determined by the box collider of the part. These colliders were be varied in size and shape to achieve a realistic simulation.

To achieve this action, a script was written and attached to the shocks and knuckle assembly. The script created a fixed joint when the colliders contact the colliders of the tool flange and delete the joint when the part collides the jig. The OnCollisionEnter event was leveraged to determine if a collision event occurred. The collision event then would pass information such as, where the collision occurred and the part which it collided with to the script. After the collision occurred, an *if* statement checked the item that collided to see if it was the end effector. If a collision occurred then at the point of collision a fixed joint was created. Then the object could be dragged by the robot like it was gripped. The same OnCollisionEnter event would be flagged when the item collides with the fixture on the table. Another *if* statement checks weather the item collided with the jig. If it did, then the part attached to the robot was scaled to zero causing it to visually disappear. This was done because originally, the *Destroy()* method was used to break the fixed joint and would cause the other joints in the robot to not work properly. When scaled to zero, the joint does not have to break, but the user still feels as though the part was placed. When the part disappears, another part was scaled up to the size in the jig. This gives the appearance the part was snapped into place to be worked on by the user. This script was designed to be easily attached to any Rigidbody the robot must grab. If the part is a Rigidbody and has a collider, the robot can grab the item. This code can be seen in Appendix H.

In this chapter, the environment and the interactions with the robot were discussed. The front knuckle assembly of a car was chosen as a spot a co-robot could be helpful and decrease production time. A scene then was devised to match Honda's manufacturing environment with some small layout changes due to the robot's working volume. The parts the robot interacts with had an attached script that would allow for the robot to pick and place them. This code was made to be versatile so the robot could interact with any object containing the script. Now the simulation was ready for testing.

Chapter 4: Testing and Evaluation

The current simulation revision was tested by running through multiple movements that users preformed. These tests were done to verify that the metric of making this revision more realistic than the previous was met. Additionally, the testing uncovered items that will be upgraded in revision three.

The Hololens was found to work well for this application, but the Hololens was not easy for new users to pick up. Hololens had only a few gestures for new users to learn to be able to pick up the Hololens and run any application. Users attempted to perform the gestures, but the dragging motion of the robot would not occur. This was due to either an incorrectly preformed gesture or the users hand was not in the sensing area. These results show the need to allow the user to use the Hololens prior to using the simulation. If the user jumps straight into the simulation, it could cause the user to think the simulation was broken and become frustrated. These first-time users had trouble dragging the robot around the scene, but experienced users would have issues dragging as well. This was because the Hololens only has a small area where gestures are recognized, When the user moved their hand outside of this area, or the head was moved outside of the area, the dragging motion was stopped. This caused the dragging to be broken up into a couple of movements to get the robot where it needed to be. Additionally, the depth of the robot was hard to control. The robot looked like it was at the correct depth, but it was hard to precisely control in the Hololens. If the user moved around they could precisely control the robot but then it would make the movements even more choppy.

The precise hand movement was needed for the simulation to work correctly. The robot still would break apart on some occasions, which the user had to avoid to continually use the simulation. The first occasion could be avoided by using a large room. When the simulation was loaded into a small room, the colliders would collide with the small real environment. This would cause the physics engine to solve the robot so that it was not colliding with the environment. These solutions were not real solutions and therefore, would cause the robot to break apart. Another situation that had to be avoided was moving the robot too quickly. If this occurred, the Hololens had trouble updating the robot quickly enough. This would cause the interpolator to fail when it must interpolate over large distances. This was reduced by decreasing lerp speed, but unfortunately was not able to be eliminated. A motion planning code should implemented to eliminate this unwanted feature.

This simulation revision updated the initial dragging feature. This feature decreased the occurrences of the robot breaking into parts, but the script used to drag the object locked the control of rotation. When dragging, the end effector's rotation was locked because there were no good ways to determine and control rotation. The Hololens' cameras picked up the wrist and hand location when an air tap occurred. The hand position would be used to determine movements in the coordinate axes. To obtain rotational data, the relative wrist, and hand location could be used. Unfortunately, there was no easy way to incorporate it into the simulation. The HandDraggable script had four options for rotation modes. The default was used which locked rotation. The remaining three options were attempted but all lock rotation in some nature.

Two of these modes can be seen in Figure 16, and 17. Adding this feature would make the simulation more realistic, and a motion planning code could eliminate the need for the rotation to be added to the dragging script.

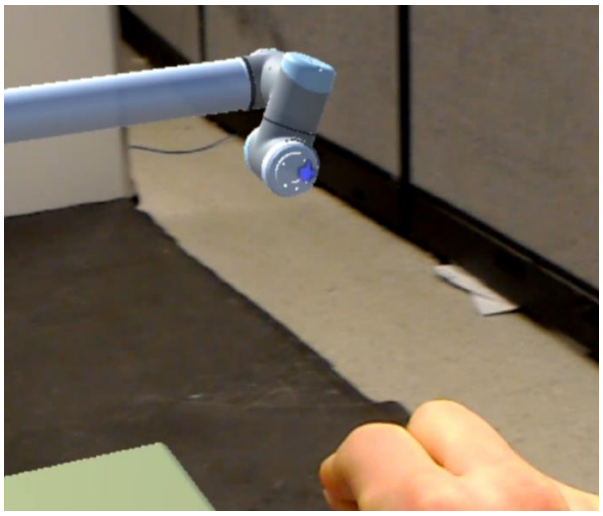

Figure 16: Default Dragging Rotation Mode

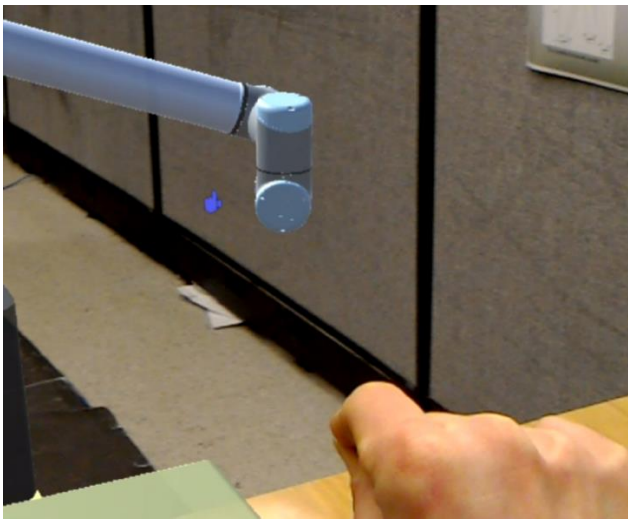

Figure 17: Orient away from User Dragging Rotation Mode

The ability to rotate the end effector when dragging became an apparent need when the snap feature was used. This feature was used for picking up objects by the robot. When items were picked up they remained in their starting positions. When the items approached the jig, they were snapped into place. This snap included a rotational snap, which was unrealistic and could be jarring to the user. Furthermore, the snap feature was reliant on the colliders size, too small and the collider was hard to hit and too large and the snapping occurs when the part is far from the jig. Additionally, the colliders used were box colliders, which are better for processing, but less accurate than a mesh collider. Since the box collider is less accurate, the parts would snap when any part of a collider hit the other. All of the snaps are unrealistic, but for this thesis they were optimized for simplicity and need for processing power. Before and after the snap can be seen in figure 18 and 19. To remove this feature, a motion planning program could be written, which would move the parts to the precise placement.

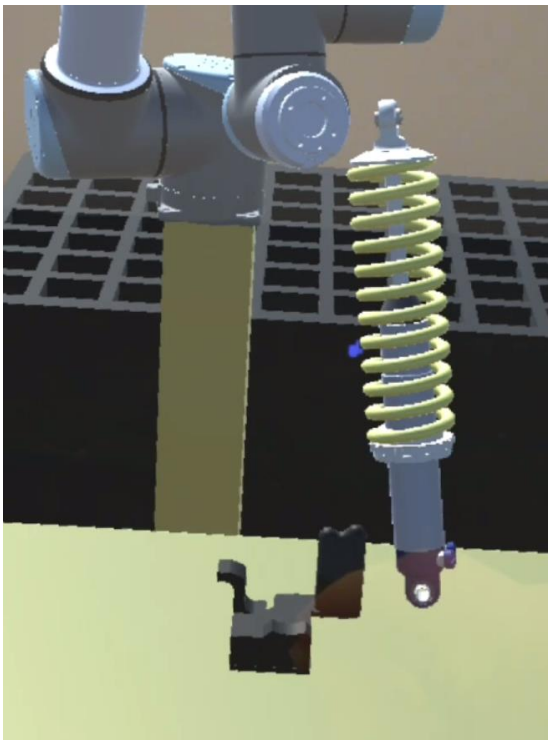

Figure 18: Grabbed Shock Before Snap

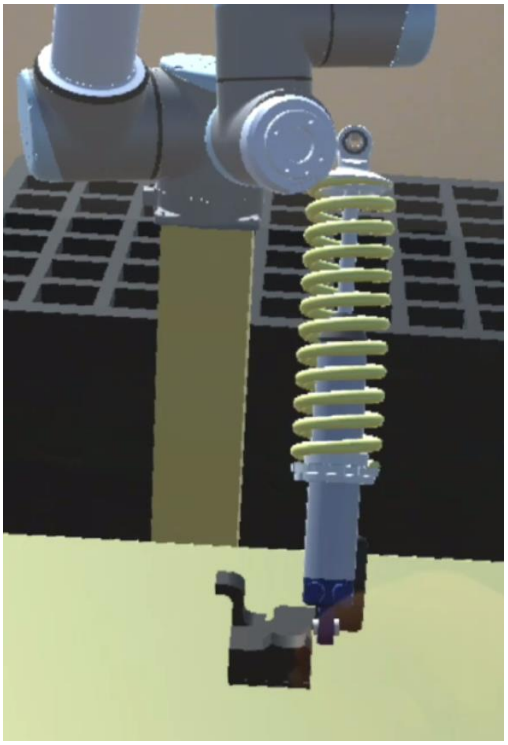

Figure 19: Snapped Shock

One last item that was identified from testing is this simulation did not incorporate a lot of human robot collaboration tasks. The focus of this thesis was making the simulation more realistic and reliable. The human placed a bolt through the assembly. This task requires fine motor skills which would be difficult to program a robot to preform. This was an ideal task for the human part of the collaboration, but the simulation only required this task once. The human did guide the robot through tasks which could mimic a real programming scenario. Human guidance through a task could make programming of co-robots quicker than conventional programming. The lack of multiple human robot collaboration tasks made this simulation a demonstration of what can be done with more time creating the simulation.

Chapter 5: Conclusion

Co-robots are the future of robotics in manufacturing. Having the ability for a human user to interact with robots will allow for virtually endless items to be manufactured. The human can take on the complex tasks while the robot complete with the lifting and repetitive tasks. To make this feasible, other technology such as augmented reality will be needed.

In this research, a demo was created to show the feasibility of using an augmented reality device to model a collaborative robot. For the demo a front car knuckle assembly was used to simulate a typical manufacturing environment with a UR-10 robot aiding the operator. Testing the device shows that Unity can support an environment with an interactive robot. Unfortunately, there was still work to be done on the simulation of the robot. Unity's hinge joints still would have issues solving positions and causing a fatal error. Moreover, it was also found that the environment used could be easily changed to mimic any manufacturing environment.

5.1 Contributions

Prior to this thesis, work with Co-robots in virtual reality had been done. These robots had been tested in differing scenarios, but only in the virtual world. This VR research aimed to make co-robots better for a medium and small-scale production. This thesis aimed to leverage new the technology to improve the research done in virtual reality. Since augmented reality layers holograms onto the real world, this device can make a simulation more realistic for the user and incorporate real world items.

This research started over the summer with two interns from the department. These two researchers set up the Hololens and figure out how to use Unity. Their research produced a simple model with a dragging robot that could place car objects from one platform to another. This was a great starting point for more research.

This thesis aimed to make the robot model more realistic and to model the robot into another more authentic manufacturing environment. This work first focused on the model of the robot. The model did not look like the real robot and would break into pieces when dragging the end effector. The colors were fixed and the breaking apart of the model was reduced. The ability to place objects on the real world was needed for a functioning AR simulation. Previous simulations with virtual reality did not need this feature because the entire simulation was virtual. After talking to Honda Associates for ideas on a manufacturing environment ideal for collaboration, a demo was produced. This demo incorporated aspects needed for multiple simulations to be created and performed. The dragging of the robot and interactions with the environment was refined. A script was written that could be attached to any object and allow for the robot to pick them up. These refinements were completed for this thesis and created a better working demo for working with co-robots.

5.2 Additional Applications and Future Work

Augmented reality has many uses in industry. This simulation could be used in a few ways which were highlighted by the Honda Associates talked to. The first of these is for training. A scenario can be programmed into the Hololens and operators can be trained on their specific location along the assembly line. They can practice going though the necessary motions to stay safe while completing their work. Additionally, the simulation can be used to increase a human operators comfort levels when working with robots. One road block for human and robot collaborating is the human's comfort level with the robot. Many workers had been told to stay away from robots that were moving for safety concerns. This idea had become engrained in workers minds that robots were unsafe when moving and working. The simulation could ease the tension without a real robot present enabling the comfort level with a robot to rise. Also, the simulation can be used to investigate where the robot should be placed for maximum efficiency. Industry can test the placements without having to shut down the entire plant. This could be a huge cost savings for companies. Having the ability to layer holograms gives the companies more feedback as compared to VR. Finally, this simulation can be used to determine safety of collaborative robots and their environment. Humans can run through the simulation and unsafe activities can be monitored. This data can help inform the placements or uses of robots in the future in different industry applications.

The simulation created for this thesis is the first stepping stone to what can be accomplished by augmented reality. The car knuckle was chosen as the first test environment for this simulation. Other environments can be explored, benefiting the use of collaborative robotics such as mounting the car battery on the assembly line. The simulation looks for the user to drag and control the robot. The functionality can be expanded to include robot motion planning to make the simulation more realistic. Multiple ended effectors should be added so that the robot can switch and complete a multitude of different tasks. One item that was investigated is using motion capture software. This would allow Unity to work with a set of cameras to determine the precise location of the hands when working in the simulation. As of now, only Hololens gestures can be used, but with this addition, any movement made by the hands can be detected allowing for a more realistic experience.

5.3 Summary

This thesis produced a working demo of co-robot and human interaction. This was done by first creating an interactive robot and placing it in a real manufacturing environment. The first revision of the interactive robot was refined. The robot was made more realistic by modifying scripts from the Virtual Reality Tool Kit. This minimized the times the robot would break apart and cause the simulation to be restarted. The human was able to interact and tell the robot where to move. A front knuckle assembly was used as the manufacturing environment and task. This environment was the starting point of many simulations that could be made. The scripts made could be easily transferred to other environments. There are two items the robot was made to grab in this demo, the shock and the knuckle. The testing showed the robot was not perfect. The physics engine could not solve the iterations fast enough and caused the robot to break apart. This thesis took steps towards making an all-around simulation that is adaptable to any manufacturing environment. In the future, simulations such as this one could make co-robots better for medium and small scale manufacturing.

Appendix A: Unity and Visual Studios Set Up

Unity 5.6.4f1 was used to create this simulation. This is not the newest version, but the free download can be found at https://unity3d.com/get-unity/download/archive. This version was found to work better with the robot model designed for this simulation. Additionally, Visual Studios 2017 was used to edit C# code and to run the app. This program can be downloaded from Microsoft for free on their website https://www.visualstudio.com/downloads/.

To set up Unity for use with the Microsoft Hololens a few steps and settings need to be set up. First when unity is opened, choose new project. This will develop the folder structure to store the assets used in the project as well as the Visual Studios solution. The Unity User Interface can be seen in Figure 15. Now that you have a new project, the Mixed Reality Tool Kit asset should be added to the project. This tool kit can be found on Github at https://github.com/Microsoft/MixedRealityToolkit and contains scripts and assets that make creating an app easy to begin.

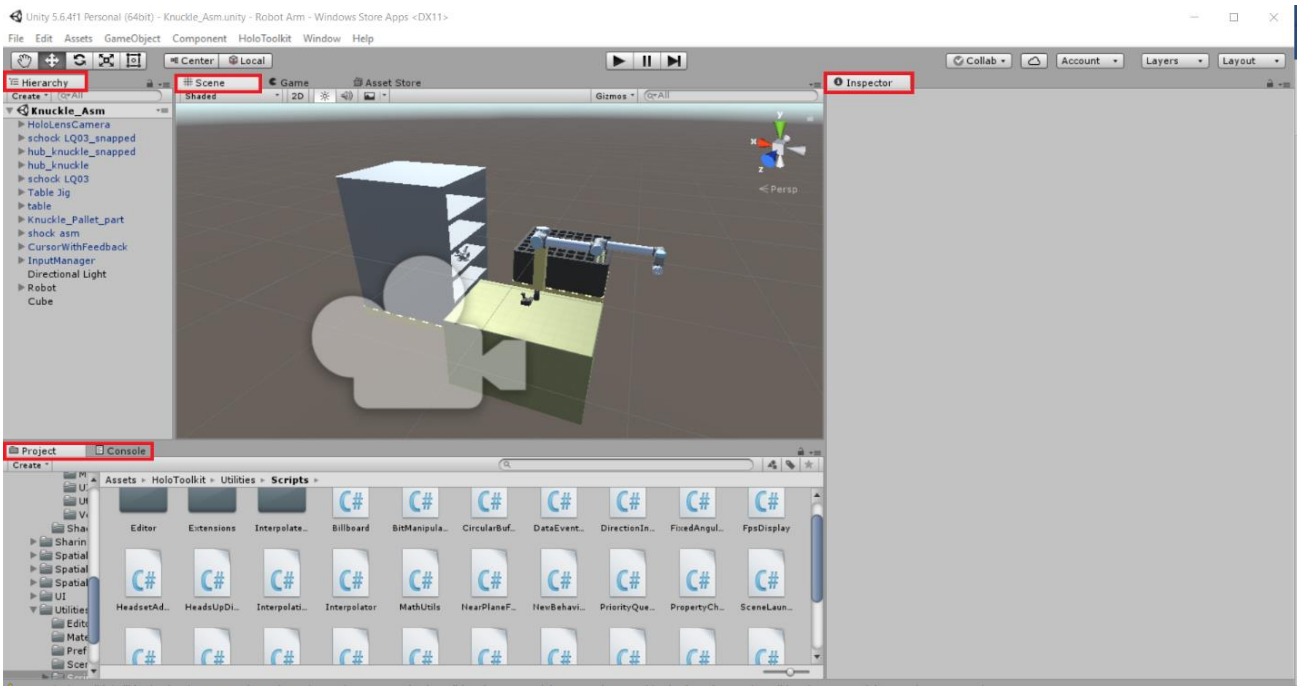

Figure 20: Unity User Interface (UI) With Key Areas Highlighted

The first setting that have to be adjusted is the player settings. These can be found in Edit→ Project Settings→Player. When clicked on the inspector tab, the windows store icon should be tapped on. Within the Other Settings tab, click on the Virtual reality supported check box. The SDK listed underneath should say Windows Holographic. Now, under the Publishing Settings→Capabilities the SpatialPerception check box needs to be checked.

The second setting that needs to be checked is the physics section. Follow Edit→ Project Settings→Physics. The default solver iteration was raised to 100 while the Default solver velocity was increased to 50. These settings allow the solver to iterate faster to give a smoother

robot when it is moving and stop the pull apart that can happen for the physics engine. Additionally, the water check marks were unchecked because they were not needed and would speed up the solver.

The third setting that needs to be changed is the quality setting. It can be found in Edit→ Project Settings→Quality. There are various quality levels that are pre-programed into Unity with the option to create your own. The “Fastest” quality level was selected to use for the simulation. Ensure that the “Fastest” level is selected. The quality can be optimized, but for now the “Fastest” level seems to work well.

To ensure that the image appears as a hologram in the environment the camera settings need to be adjusted. In the Mixed Reality Tool Kit, a camera was made for the Hololens called HoloLensCamera. Drag this into the Hierarchy tree and delete the other camera. If one does not want to use this camera, then the main camera can be used with two modifications. Under Camera in the Inspector change the Background setting to black (0,0,0) and clear flags to solid color. This will make the camera appear as it should on the Hololens.

The build settings should be modified to fit the Figure 16. To add the open scene click the Add Open Scene button. Then choose the Windows Store. If Unity was just installed, then an add-in must be istalled to use Unity for the Hololens. The SDK should be set for Universal 10 and the target device set to the hololens. Finally the UWP Build type should be D3D with Unity C# Projects checked. When Build is pressed, Unity will compile all items needed to create a Visual Studios solution and open the folder with this application just coded in it.

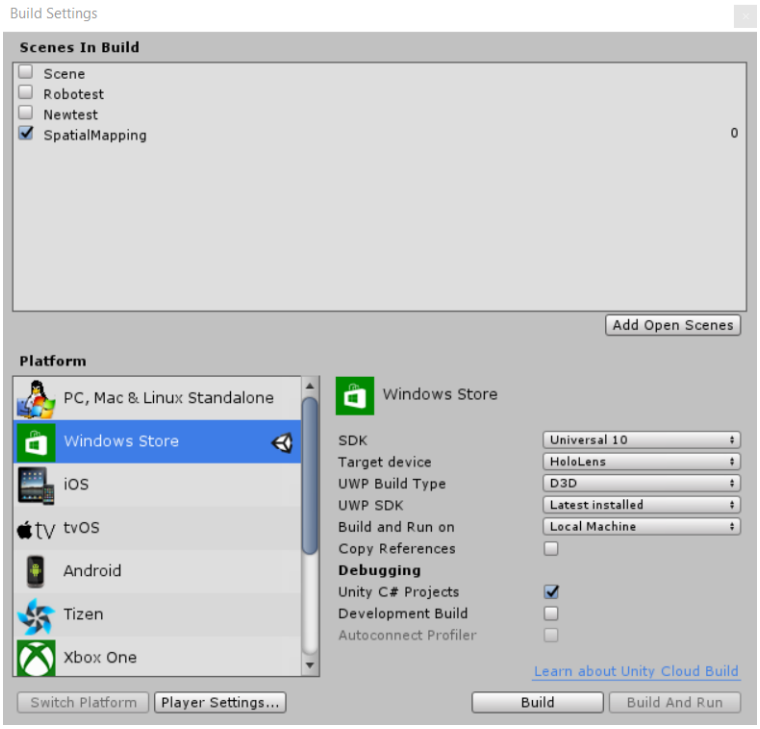

Figure 21: Build Settings Window

To run the app, open the solution in Visual Studios. The two dropdown boxes must be set to Release and x86. This ensures that the solution runs on the Hololens. The next dropdown shuld be set on device if the Hololens is connected via USB, or Remote Machine if the app will bedeployed over wifi. These settings can be seen in Figure 17. Finally, go to Debug→Start Without Debugging to deploy the app. Visual Studios will take a second to deploy, and it may gve an error the first time. The error disapears when run the second time.

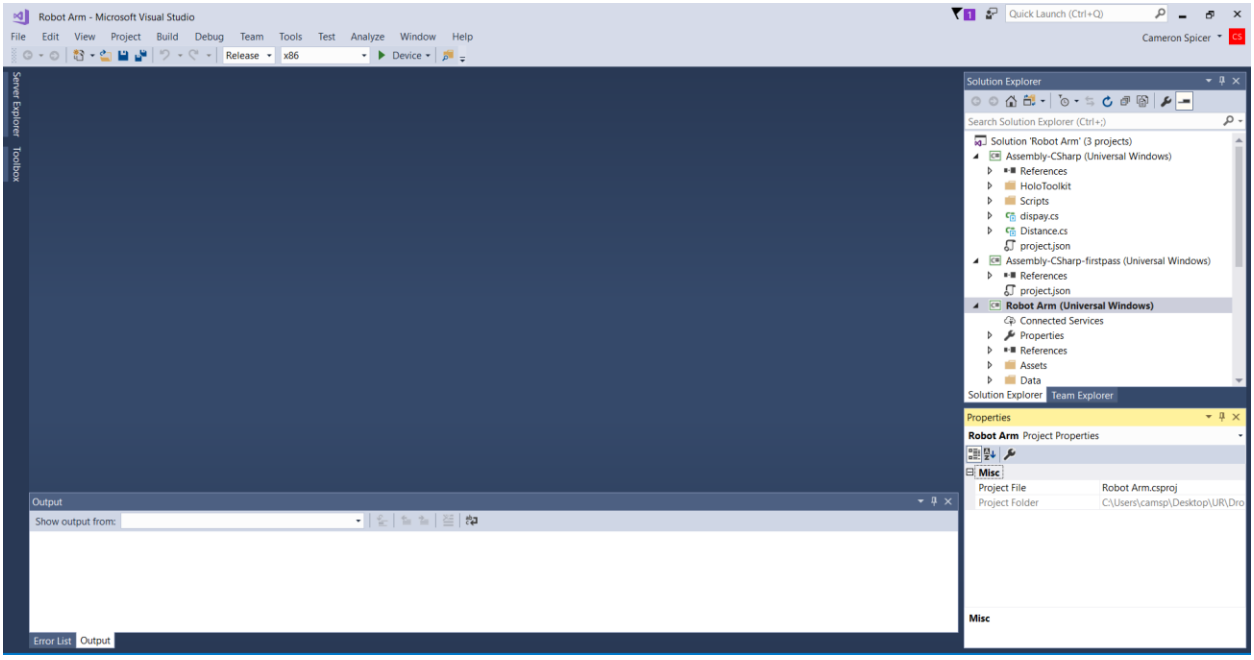

Figure 22: Visual Studios Interface Ready to Run the App

Appendix B: Creating Custom Objects in Unity

Unity provides a list of objects that can be added to the game without making and importing them. These include a cube, sphere and other simple geometries. These can be limiting in what one can make, so the creation of custom objects is ideal.

To start the process of making a game object, first one must be made or found online. GrabCAD is a powerful website where people share their CAD for almost anything, but there are a lot of file types on GrabCAD that Unity can't handle. If an object is created from scratch then specific file types should be used to interface with Unity. Since SOLIDWORKS is the main 3D modeler used at OSU, this is a good starting point. In SOLIDWORKS the model is created using the correct units and noted for later use. The SOLIDWORKS part file and STEP files should be saved. An easy way to do this is by using the SimMechanics export tool in SOLIDWORKS for an entire assembly.

The STEP file is then imported into 3DSMax. This software is an AutoDesk product which offers other file types that SOLIDWORKS does not. While in 3DSMax, texture and color should be applied to give the part the correct look. The SOLIDWORKS color will be stripped when exported from 3DSMax. The STEP file does not retain dimensional information which is required for import into Unity. As the workspace is defined in meters, and unity is also in meters, it would be sensible that there is direct compatibility. However, the default file formats of 3D objects is in units of inches before the import. It is necessary then to rescale the object in Unity by a factor of 0.0254. This is done in the 3DSMax settings before export. Then on export to FBX file the correct unit of meters will appear in Unity. To export the file go to File→Export and export as a FBX. This file type allows for the measurement system to be saved so the hologram is the correct size in Unity.

The final step is to import the Asset into Unity. This is done by finding the .FBX file and dragging it into the project panel on the bottom of the Unity interface.

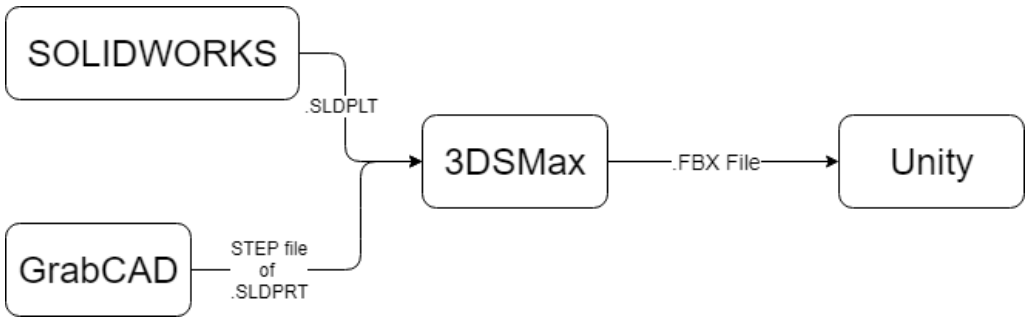

Figure 23: Flow Chart for Creating Unity Parts

Appendix C: Gestures in Microsoft Hololens

The Hololens has a few gestures built-in and easy to use in Unity. The first of these is the Bloom Gesture. This gesture always takes the user to the start menu. This is reserved for the Hololens to use. To preform the gesture, raise all five fingers together and then open them to a flat hand.

Figure 24: Closed Bloom Gesture

Figure 25: Open Bloom Gesture

The air tap gesture is the main gesture used in this thesis. It is like clicking a mouse button. One finger is raised then lowered back into the fist. This will register as a tap and can be used to interact with the environment. The Tap and Hold is a variation of this where the user can hold onto the object. Other gestures and for more descriptions of these gestures can be found at https://support.microsoft.com/en-us/help/12644/hololens-use-gestures.

Figure 26: Air Tap Gesture

Appendix D: Windows Device Portal

The Windows Device Portal comes standard with the Hololens and has some adventitious features. To connect to the device first plug it into the computer over USB and turn on the Hololens. Next, go to a browser and enter http://127.0.0.1:10080 into the bar. Now a sign in box will show up. Once entered it will take you to the splash page.

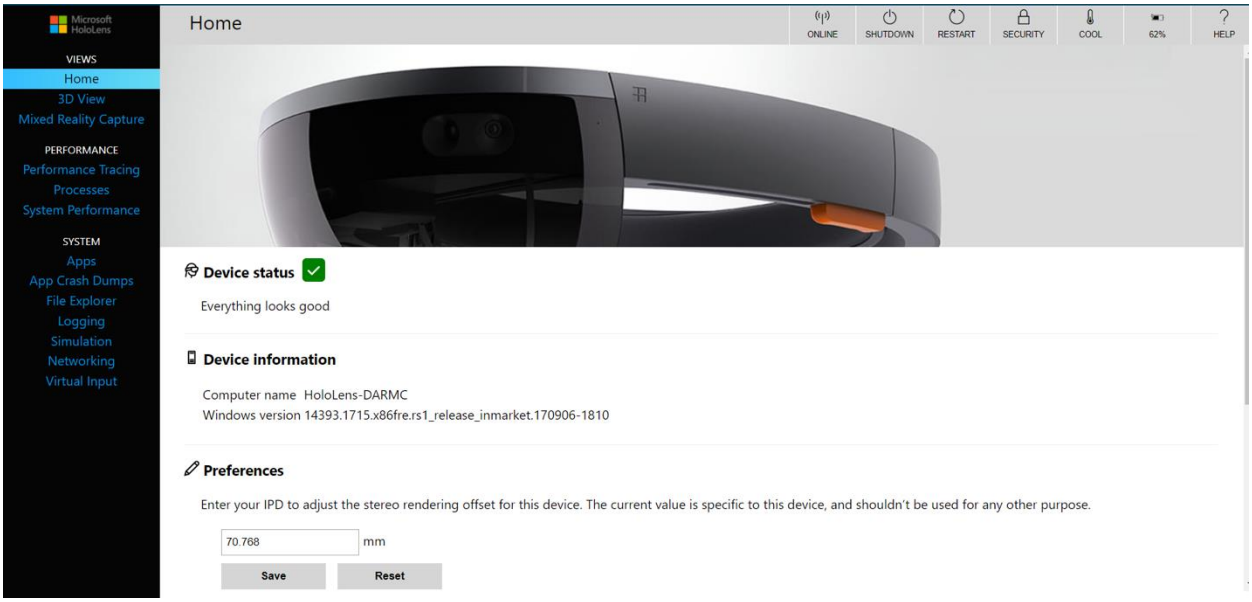

Figure 27: Splash Page of the Windows Device Portal

From the device portal a few things can be achieved. The first is the ability to see in real time the environment that the Hololens sees. One can take a rendering of the environment and save it for later. Additionally, the performance of the device can be tracked in this portal in real time. This portal allows for easy management of the apps on the device and network settings that can be changed. Finally, the most important feature that the portal has is the ability to capture images and videos. On this tab a view of the picture that is to be taken can be seen as well as some options to change what is seen in the picture or video. Once the video or picture is captured, the portal allows for a simple download of the images. This is handy to show others what is being seen by the user. For more information on the portal or its settings and uses please visit:

https://developer.microsoft.com/en-us/windows/mixed-reality/using_the_windows_device_portal

Appendix E: Modified Hand Draggable Script

```
using UnityEngine;
using System;

namespace HoloToolkit.Unity.InputModule
{
        /// <summary>
        /// Component that allows dragging an object with your hand on HoloLens.
        /// Dragging is done by calculating the angular delta and z-delta between the current and previous
        hand positions,
        /// and then repositioning the object based on that.
        /// </summary>
        public class HandDraggableV4 : MonoBehaviour,
        IFocusable,
        IInputHandler,
        ISourceStateHandler
        {
        /// <summary>
        /// Event triggered when dragging starts.
        /// </summary>
        public event Action StartedDragging;
        /// <summary>
        /// Event triggered when dragging stops.
        /// </summary>
        public event Action StoppedDragging;
        Vector3 temppos;
        Vector3 Lastpos;
        [Tooltip("Transform that will be dragged. Defaults to the object of the component.")]
        public Transform HostTransform;
        [Tooltip("Scale by which hand movement in z is multipled to move the dragged object.")]
        public float DistanceScale = 2f;
        public Rigidbody RB;
        public enum RotationModeEnum
        {
                Default,
                LockObjectRotation,
                OrientTowardUser,
                OrientTowardUserAndKeepUpright
        }
        public RotationModeEnum RotationMode = RotationModeEnum.Default;
        [Tooltip("Controls the speed at which the object will interpolate toward the desired position")]
        [Range(0.01f, 1.0f)]
        public float PositionLerpSpeed = 0.2f;
        [Tooltip("Controls the speed at which the object will interpolate toward the desired rotation")]
        [Range(0.01f, 1.0f)]
        public float RotationLerpSpeed = 0.2f;
        public bool IsDraggingEnabled = true;
        private Camera mainCamera;
        private bool isDragging;
        private bool isGazed;
        private Vector3 objRefForward;
        private Vector3 objRefUp;
        private float objRefDistance;
        private Quaternion gazeAngularOffset;
        private float handRefDistance;
        private Vector3 objRefGrabPoint;
        public float maxdist = 1.27f;
        private Vector3 draggingPosition;
        private Quaternion draggingRotation;
        private IInputSource currentInputSource = null;
        private uint currentInputSourceId;
        public GameObject ToolFlange;
        public GameObject Shoulder;
        public Rigidbody rbUA;
        public Rigidbody rbS;
        public Rigidbody rbLA;
```

```
        public Rigidbody rbW1;
        public Rigidbody rbW2;
private void Start()
        {
                if (HostTransform == null)
                {
                        HostTransform = transform;
                }
                mainCamera = Camera.main;
                ToolFlange = GameObject.Find("Tool Flange");
                Shoulder = GameObject.Find("Shoulder");
                Lastpos = ToolFlange.transform.position;
                rbUA = GameObject.Find("Upper Arm").GetComponent<Rigidbody>();
                rbLA = GameObject.Find("Lower Arm").GetComponent<Rigidbody>();
                rbS = GameObject.Find("Shoulder").GetComponent<Rigidbody>();
                rbW2 = GameObject.Find("Wrist 2").GetComponent<Rigidbody>();
                rbW1 = GameObject.Find("Wrist 1 (1)").GetComponent<Rigidbody>();
        }
private void OnDestroy()
        {
                if (isDragging)
                {
                        StopDragging();
                }
                if (isGazed)
                {
                        OnFocusExit();
                }
        }
private void ReturnKinematic()
        {
            //switches the robot between kinematic and not when dragging
            rbUA.isKinematic = true;
            rbLA.isKinematic = true;
            rbS.isKinematic = true;
            rbW1.isKinematic = true;
            rbW2.isKinematic = true;
        }
private void length()
        {
               //checks distance between the tool flange and shoulder
                float dista = Vector3.Distance(ToolFlange.transform.position,
                Shoulder.transform.position);
                Debug.Log(string.Format("Dist: {0}", dista));
                if (dista >= maxdist)
                {
                        //jumps to last position that was inside the working area
                        ToolFlange.transform.position = Lastpos;
                        StopDragging();
                }
                else{ return;}
        }
private void Update()
        {
                length();//checks on each iteration
                if (IsDraggingEnabled && isDragging)
                {
                        Lastpos = ToolFlange.transform.position;
                        rbUA.isKinematic = false;
                        rbLA.isKinematic = false;
                        rbW1.isKinematic = false;
                        rbW2.isKinematic = false;
                        rbS.isKinematic = false;
                        UpdateDragging();
                }

        }
```

```
                /// <summary>
                /// Starts dragging the object.
                /// </summary>
public void StartDragging()
        {
                if (!IsDraggingEnabled)
                {
                        return;
                }

                if (isDragging)
                {
                        return;
                }

                // Add self as a modal input handler, to get all inputs during the manipulation
                InputManager.Instance.PushModalInputHandler(gameObject);
                isDragging = true;
                //GazeCursor.Instance.SetState(GazeCursor.State.Move);
                //GazeCursor.Instance.SetTargetObject(HostTransform);
                Vector3 gazeHitPosition = GazeManager.Instance.HitInfo.point;
                Vector3 handPosition;
                currentInputSource.TryGetPosition(currentInputSourceId, out handPosition);
                Vector3 pivotPosition = GetHandPivotPosition();
                handRefDistance = Vector3.Magnitude(handPosition - pivotPosition);
                objRefDistance = Vector3.Magnitude(gazeHitPosition - pivotPosition);
                Vector3 objForward = HostTransform.forward;
                Vector3 objUp = HostTransform.up;
                // Store where the object was grabbed from
                objRefGrabPoint = mainCamera.transform.InverseTransformDirection(HostTransform.position -
                gazeHitPosition);
                Vector3 objDirection = Vector3.Normalize(gazeHitPosition - pivotPosition);
                Vector3 handDirection = Vector3.Normalize(handPosition - pivotPosition);
                objForward = mainCamera.transform.InverseTransformDirection(objForward);
                // in camera space
                objUp = mainCamera.transform.InverseTransformDirection(objUp);
                // in camera space
                objDirection = mainCamera.transform.InverseTransformDirection(objDirection);
                // in camera space
                handDirection = mainCamera.transform.InverseTransformDirection(handDirection);
                // in camera space
                objRefForward = objForward;
                objRefUp = objUp;
                // Store the initial offset between the hand and the object, so that we can consider it
                when dragging
                gazeAngularOffset = Quaternion.FromToRotation(handDirection, objDirection);
                draggingPosition = gazeHitPosition;
                StartedDragging.RaiseEvent();
        }

        /// <summary>
        /// Gets the pivot position for the hand, which is approximated to the base of the neck.
        /// </summary>          /// <returns>Pivot position for the hand.</returns>
private Vector3 GetHandPivotPosition()
                {
                        Vector3 pivot = Camera.main.transform.position + new Vector3(0, -0.2f, 0) -
                        Camera.main.transform.forward * 0.2f; // a bit lower and behind
                        return pivot;
                }

        /// <summary>
        /// Enables or disables dragging.
        /// </summary>
        /// <param name="isEnabled">Indicates whether dragging should be enabled or disabled.</param>
```

```
public void SetDragging(bool isEnabled)
        {
                if (IsDraggingEnabled == isEnabled)
                {
                        return;
                }

                IsDraggingEnabled = isEnabled;
                if (isDragging)
                {
                        StopDragging();
                }
        }

        /// <summary>
        /// Update the position of the object being dragged.
        /// </summary>
private void UpdateDragging()
        {
                Vector3 newHandPosition;
                currentInputSource.TryGetPosition(currentInputSourceId, out newHandPosition);
                Vector3 pivotPosition = GetHandPivotPosition();
                Vector3 newHandDirection = Vector3.Normalize(newHandPosition - pivotPosition);
                newHandDirection = mainCamera.transform.InverseTransformDirection(newHandDirection);
                // in camera space
                Vector3 targetDirection = Vector3.Normalize(gazeAngularOffset * newHandDirection);
                targetDirection = mainCamera.transform.TransformDirection(targetDirection);
                // back to world space
                float currenthandDistance = Vector3.Magnitude(newHandPosition - pivotPosition);
                float distanceRatio = currenthandDistance / handRefDistance;
                float distanceOffset = distanceRatio > 0 ? (distanceRatio - 1f) * DistanceScale : 0;
                float targetDistance = objRefDistance + distanceOffset;
                draggingPosition = pivotPosition + (targetDirection * targetDistance);
                if (RotationMode == RotationModeEnum.OrientTowardUser || RotationMode ==
                RotationModeEnum.OrientTowardUserAndKeepUpright)
                {
                        draggingRotation = Quaternion.LookRotation(HostTransform.position -
                        pivotPosition);
                }
                else if (RotationMode == RotationModeEnum.LockObjectRotation)
                {
                        draggingRotation = HostTransform.rotation;
                }
                else // RotationModeEnum.Default
                {
                        Vector3 objForward = mainCamera.transform.TransformDirection(objRefForward);
                        // in world space
                        Vector3 objUp = mainCamera.transform.TransformDirection(objRefUp);
                        // in world space
                        draggingRotation = Quaternion.LookRotation(objForward, objUp);
                }

                // Apply Final Position
                HostTransform.position = Vector3.Lerp(HostTransform.position, draggingPosition +
                mainCamera.transform.TransformDirection(objRefGrabPoint), PositionLerpSpeed);
                // Apply Final Rotation
                HostTransform.rotation = Quaternion.Lerp(HostTransform.rotation, draggingRotation,
                RotationLerpSpeed);
                if (RotationMode == RotationModeEnum.OrientTowardUserAndKeepUpright)
                {
                        Quaternion upRotation = Quaternion.FromToRotation(HostTransform.up, Vector3.up);
                                HostTransform.rotation = upRotation * HostTransform.rotation;
                }

        }
```

```
        /// <summary>
        /// Stops dragging the object.
        /// </summary>
public void StopDragging()
        {
                if (!isDragging)
                {
                        return;
                }

                // Remove self as a modal input handler
                InputManager.Instance.PopModalInputHandler();
                isDragging = false;
                currentInputSource = null;
                ReturnKinematic();
                StoppedDragging.RaiseEvent();
        }

public void OnFocusEnter()
        {
                if (!IsDraggingEnabled)
                {
                        return;
                }

                if (isGazed)
                {
                        return;
                }

                        isGazed = true;
        }
public void OnFocusExit()
        {
                if (!IsDraggingEnabled)
                {
                        return;
                }

                if (!isGazed)
                {
                        return;
                }

                        isGazed = false;
        }

public void OnInputUp(InputEventData eventData)
        {
                if (currentInputSource != null &&
                eventData.SourceId == currentInputSourceId)
                {
                        StopDragging();
                }
        }

public void OnInputDown(InputEventData eventData)
        {
                if (isDragging)
                {
                        // We're already handling drag input, so we can't start a new drag operation.
                        return;
                }

                if (!eventData.InputSource.SupportsInputInfo(eventData.SourceId,
                SupportedInputInfo.Position))
                {
                        // The input source must provide positional data for this script to be usable
                        return;
```

```
                }

                currentInputSource = eventData.InputSource;
                currentInputSourceId = eventData.SourceId;
                StartDragging();
        }

public void OnSourceDetected(SourceStateEventData eventData)
        {
                // Nothing to do
        }

public void OnSourceLost(SourceStateEventData eventData)
        {
                if (currentInputSource != null && eventData.SourceId == currentInputSourceId)
                {
                        StopDragging();
                }
        }

    }
}
```

Appendix F: Modified Tap To Place Script

```
using System.Collections.Generic;
using UnityEngine;
using HoloToolkit.Unity.SpatialMapping;

namespace HoloToolkit.Unity.InputModule
{
    /// <summary>
    /// The TapToPlace class is a basic way to enable users to move objects
    /// and place them on real world surfaces.
    /// Put this script on the object you want to be able to move.
    /// Users will be able to tap objects, gaze elsewhere, and perform the tap gesture again to place.
    /// This script is used in conjunction with GazeManager, WorldAnchorManager, and
        SpatialMappingManager.
    /// </summary>
    [RequireComponent(typeof(Collider))]
    [RequireComponent(typeof(Interpolator))]
    public class TapToPlace : MonoBehaviour, IInputClickHandler
    {
        [Tooltip("Distance from camera to keep the object while placing it.")]
        public float DefaultGazeDistance = 2.0f;
        [Tooltip("Supply a friendly name for the anchor as the key name for the WorldAnchorStore.")]
        public string SavedAnchorFriendlyName = "SavedAnchorFriendlyName";
        [Tooltip("Place parent on tap instead of current game object.")]
        public bool PlaceParentOnTap;
        [Tooltip("Specify the parent game object to be moved on tap, if the immediate parent is not
        desired.")]
        public GameObject ParentGameObjectToPlace;
        /// <summary>
        /// Keeps track of if the user is moving the object or not.
        /// Setting this to true will enable the user to move and place the object in the scene.
        /// Useful when you want to place an object immediately.
        /// </summary>
        [Tooltip("Setting this to true will enable the user to move and place the object in the scene
        without needing to tap on the object. Useful when you want to place an object immediately.")]
        public bool IsBeingPlaced;
        [Tooltip("Setting this to true will allow this behavior to control the DrawMesh property on the
        spatial mapping.")]
        public bool AllowMeshVisualizationControl = true;
        /// <summary>
        /// The default ignore raycast layer built into unity.
        /// </summary>
        private const int IgnoreRaycastLayer = 2;
        private Interpolator interpolator;
        public Rigidbody rbUA;
        public Rigidbody rbS;
        public Rigidbody rbLA;
        public Rigidbody rbW1;
        public Rigidbody rbW2;
        private static Dictionary<GameObject, int> defaultLayersCache = new Dictionary<GameObject, int>();
protected virtual void Start()
        {
            // Make sure we have all the components in the scene we need.
            if (WorldAnchorManager.Instance == null)
            {
                Debug.LogError("This script expects that you have a WorldAnchorManager component in your
                scene.");
            }

            if (WorldAnchorManager.Instance != null)
            {
                // If we are not starting out with actively placing the object, give it a World Anchor
                if (!IsBeingPlaced)
                {
                    WorldAnchorManager.Instance.AttachAnchor(gameObject, SavedAnchorFriendlyName);
```

```
            }
        }

        DetermineParent();
        interpolator = PlaceParentOnTap
            ? ParentGameObjectToPlace.EnsureComponent<Interpolator>()
            : gameObject.EnsureComponent<Interpolator>();

        if (IsBeingPlaced)
        {
            HandlePlacement();
        }
        rbUA = GameObject.Find("Upper Arm").GetComponent<Rigidbody>();
        rbLA = GameObject.Find("Lower Arm").GetComponent<Rigidbody>();
        rbS = GameObject.Find("Shoulder").GetComponent<Rigidbody>();
        rbW2 = GameObject.Find("Wrist 2").GetComponent<Rigidbody>();
        rbW1 = GameObject.Find("Wrist 1 (1)").GetComponent<Rigidbody>();
    }
protected virtual void Update()
    {
        if (!IsBeingPlaced)
        {

            //GameObject.Find("table").transform.localScale = new Vector3(0, 0, 0);
            //add kinematic=false and table diapears
            return;
        }
        else
        {

            //GameObject.Find("table").transform.localScale = new Vector3(1, 1, 1);
        }

        Vector3 headPosition = Camera.main.transform.position;
        Vector3 gazeDirection = Camera.main.transform.forward;
        // If we're using the spatial mapping, check to see if we got a hit, else use the gaze
        position.
        RaycastHit hitInfo;
        Vector3 placementPosition = SpatialMappingManager.Instance != null &&
        Physics.Raycast(headPosition, gazeDirection, out hitInfo, 30.0f,
        SpatialMappingManager.Instance.LayerMask)
                ? hitInfo.point
                : (GazeManager.Instance.HitObject == null
                    ? GazeManager.Instance.GazeOrigin + GazeManager.Instance.GazeNormal *
                    DefaultGazeDistance
                    : GazeManager.Instance.HitPosition);

        // Here is where you might consider adding intelligence
        // to how the object is placed.  For example, consider
        // placing based on the bottom of the object's
        // collider so it sits properly on surfaces.

        if (PlaceParentOnTap)
        {
            placementPosition = ParentGameObjectToPlace.transform.position + (placementPosition -
            gameObject.transform.position);
        }

        // update the placement to match the user's gaze.
        interpolator.SetTargetPosition(placementPosition);

        // Rotate this object to face the user.
        interpolator.SetTargetRotation(Quaternion.Euler(0, Camera.main.transform.localEulerAngles.y,
        0));
    }
public virtual void OnInputClicked(InputClickedEventData eventData)
    {
        // On each tap gesture, toggle whether the user is in placing mode.
        IsBeingPlaced = !IsBeingPlaced;
```

```
            if (rbUA.isKinematic == false)
            {
                rbUA.isKinematic = true;
                rbLA.isKinematic = true;
                rbS.isKinematic = true;
                rbW1.isKinematic = true;
                rbW2.isKinematic = true;
            }

            HandlePlacement();
        }

        private void HandlePlacement()
        {
            if (IsBeingPlaced)
            {
                SetLayerRecursively(transform, useDefaultLayer: false);
                InputManager.Instance.AddGlobalListener(gameObject);

                // If the user is in placing mode, display the spatial mapping mesh.
                if (AllowMeshVisualizationControl)
                {
                    SpatialMappingManager.Instance.DrawVisualMeshes = true;
                }
#if UNITY_WSA && !UNITY_EDITOR

                //Removes existing world anchor if any exist.
                WorldAnchorManager.Instance.RemoveAnchor(gameObject);
#endif
            }
            else
            {
                SetLayerRecursively(transform, useDefaultLayer: true);
                // Clear our cache in case we added or removed gameobjects between taps
                defaultLayersCache.Clear();
                InputManager.Instance.RemoveGlobalListener(gameObject);

                // If the user is not in placing mode, hide the spatial mapping mesh.
                if (AllowMeshVisualizationControl)
                {
                    SpatialMappingManager.Instance.DrawVisualMeshes = false;
                }
#if UNITY_WSA && !UNITY_EDITOR

                // Add world anchor when object placement is done.
                WorldAnchorManager.Instance.AttachAnchor(gameObject, SavedAnchorFriendlyName);
#endif
            }
        }

        private void DetermineParent()
        {
            if (!PlaceParentOnTap) { return; }

            if (ParentGameObjectToPlace == null)
            {
                if (gameObject.transform.parent == null)
                {
                    Debug.LogWarning("The selected GameObject has no parent.");
                    PlaceParentOnTap = false;
                }
                else
                {
                    Debug.LogWarning("No parent specified. Using immediate parent instead: " +
                    gameObject.transform.parent.gameObject.name);
                    ParentGameObjectToPlace = gameObject.transform.parent.gameObject;
                }
            }
```

```
            if (ParentGameObjectToPlace != null &&
            !gameObject.transform.IsChildOf(ParentGameObjectToPlace.transform))
            {
                Debug.LogWarning("The specified parent object is not a parent of this object.");
            }
        }
private static void SetLayerRecursively(Transform objectToSet, bool useDefaultLayer)
        {
            if (useDefaultLayer)
            {
                int defaultLayerId;
                if (defaultLayersCache.TryGetValue(objectToSet.gameObject, out defaultLayerId))
                {
                    objectToSet.gameObject.layer = defaultLayerId;
                    defaultLayersCache.Remove(objectToSet.gameObject);
                }
            }
            else
            {
                defaultLayersCache.Add(objectToSet.gameObject, objectToSet.gameObject.layer);

                objectToSet.gameObject.layer = IgnoreRaycastLayer;
            }

            for (int i = 0; i < objectToSet.childCount; i++)
            {
                SetLayerRecursively(objectToSet.GetChild(i), useDefaultLayer);
            }
        }
    }
}
```

Appendix G: Modified Interpolator Script

```
using UnityEngine;
namespace HoloToolkit.Unity
{
    /// <summary>
    /// A MonoBehaviour that interpolates a transform's position, rotation or scale.
    /// </summary>
    public class Interpolator : MonoBehaviour
    {
        [Tooltip("When interpolating, use unscaled time. This is useful for games that have a pause
        mechanism or otherwise adjust the game timescale.")]
        public bool UseUnscaledTime = true;
        // A very small number that is used in determining if the Interpolator
        // needs to run at all.
        private const float smallNumber = 0.0000001f;
        // The movement speed in meters per second
        public float PositionPerSecond = 30.0f;
        // The rotation speed, in degrees per second
        public float RotationDegreesPerSecond = 720.0f;
        // Adjusts rotation speed based on angular distance
        public float RotationSpeedScaler = 0.0f;
        // The amount to scale per second
        public float ScalePerSecond = 5.0f;
        //Forces the object that is to be moved
        public string MovingObj = "robot";
        // Lerp the estimated targets towards the object each update,
        // slowing and smoothing movement.
        [HideInInspector]
        public bool SmoothLerpToTarget = false;
        [HideInInspector]
        public float SmoothPositionLerpRatio = 0.5f;
        [HideInInspector]
        public float SmoothRotationLerpRatio = 0.5f;
        [HideInInspector]
        public float SmoothScaleLerpRatio = 0.5f;
        // Position data
        private Vector3 targetPosition;
        /// <summary>
        /// True if the transform's position is animating; false otherwise.
        /// </summary>
        public bool AnimatingPosition { get; private set; }
        // Rotation data
        private Quaternion targetRotation;
        /// <summary>
        /// True if the transform's rotation is animating; false otherwise.
        /// </summary>
        public bool AnimatingRotation { get; private set; }
        // Local Rotation data
        private Quaternion targetLocalRotation;
        /// <summary>
        /// True if the transform's local rotation is animating; false otherwise.
        /// </summary>
        public bool AnimatingLocalRotation { get; private set; }
        // Scale data
        private Vector3 targetLocalScale;
        /// <summary>
        /// True if the transform's scale is animating; false otherwise.
        /// </summary>
        public bool AnimatingLocalScale { get; private set; }
        /// <summary>
        /// The event fired when an Interpolation is started.
        /// </summary>
        public event System.Action InterpolationStarted;
        /// <summary>
        /// The event fired when an Interpolation is completed.
```

```
        /// </summary>
        public event System.Action InterpolationDone;
        /// <summary>
        /// The velocity of a transform whose position is being interpolated.
        /// </summary>
        public Vector3 PositionVelocity { get; private set; }
        private Vector3 oldPosition = Vector3.zero;
        /// <summary>
        /// True if position, rotation or scale are animating; false otherwise.
        /// </summary>
        public bool Running
        {
            get
            {
                return (AnimatingPosition || AnimatingRotation || AnimatingLocalRotation ||
                AnimatingLocalScale);
            }
        }
public void Awake()
        {
            targetPosition = transform.position;
            targetRotation = transform.rotation;
            targetLocalRotation = transform.localRotation;
            targetLocalScale = transform.localScale;

            enabled = false;
        }

        /// <summary>
        /// Sets the target position for the transform and if position wasn't
        /// already animating, fires the InterpolationStarted event.
        /// </summary>
        /// <param name="target">The new target position to for the transform.</param>
public void SetTargetPosition(Vector3 target)
        {
            bool wasRunning = Running;

            targetPosition = target;

            float magsq = (targetPosition - transform.position).sqrMagnitude;
            if (magsq > smallNumber)
            {
                AnimatingPosition = true;
                enabled = true;

                if (InterpolationStarted != null && !wasRunning)
                {
                    InterpolationStarted();
                }
            }
            else
            {
                // Set immediately to prevent accumulation of error.
                GameObject.Find(MovingObj).transform.position = target;
                AnimatingPosition = false;
            }
        }

        /// <summary>
        /// Sets the target rotation for the transform and if rotation wasn't
        /// already animating, fires the InterpolationStarted event.
        /// </summary>
        /// <param name="target">The new target rotation for the transform.</param>
public void SetTargetRotation(Quaternion target)
        {
            bool wasRunning = Running;

            targetRotation = target;
```

```
            if (Quaternion.Dot(transform.rotation, target) < 1.0f)
            {
                AnimatingRotation = true;
                enabled = true;

                if (InterpolationStarted != null && !wasRunning)
                {
                    InterpolationStarted();
                }
            }
            else
            {
                // Set immediately to prevent accumulation of error.
                GameObject.Find(MovingObj).transform.rotation = target;
                AnimatingRotation = false;
            }
        }

        /// <summary>
        /// Sets the target local rotation for the transform and if rotation
        /// wasn't already animating, fires the InterpolationStarted event.
        /// </summary>
        /// <param name="target">The new target local rotation for the transform.</param>
public void SetTargetLocalRotation(Quaternion target)
        {
            bool wasRunning = Running;

            targetLocalRotation = target;

            if (Quaternion.Dot(transform.localRotation, target) < 1.0f)
            {
                AnimatingLocalRotation = true;
                enabled = true;

                if (InterpolationStarted != null && !wasRunning)
                {
                    InterpolationStarted();
                }
            }
            else
            {
                // Set immediately to prevent accumulation of error.
                transform.localRotation = target;
                AnimatingLocalRotation = false;
            }
        }

        /// <summary>
        /// Sets the target local scale for the transform and if scale
        /// wasn't already animating, fires the InterpolationStarted event.
        /// </summary>
        /// <param name="target">The new target local rotation for the transform.</param>
public void SetTargetLocalScale(Vector3 target)
        {
            bool wasRunning = Running;

            targetLocalScale = target;

            float magsq = (targetLocalScale - transform.localScale).sqrMagnitude;
            if (magsq > Mathf.Epsilon)
            {
                AnimatingLocalScale = true;
                enabled = true;

                if (InterpolationStarted != null && !wasRunning)
                {
                    InterpolationStarted();
                }
            }
```

```
            else
            {
                // set immediately to prevent accumulation of error
                transform.localScale = target;
                AnimatingLocalScale = false;
            }
        }

        /// <summary>
        /// Interpolates smoothly to a target position.
        /// </summary>
        /// <param name="start">The starting position.</param>
        /// <param name="target">The destination position.</param>
        /// <param name="deltaTime">Caller-provided Time.deltaTime.</param>
        /// <param name="speed">The speed to apply to the interpolation.</param>
        /// <returns>New interpolated position closer to target</returns>
public static Vector3 NonLinearInterpolateTo(Vector3 start, Vector3 target, float deltaTime, float
        speed)
        {
            // If no interpolation speed, jump to target value.
            if (speed <= 0.0f)
            {
                return target;
            }
            Vector3 distance = (target - start);
            // When close enough, jump to the target
            if (distance.sqrMagnitude <= Mathf.Epsilon)
            {
                return target;
            }
            // Apply the delta, then clamp so we don't overshoot the target
            Vector3 deltaMove = distance * Mathf.Clamp(deltaTime * speed, 0.0f, 1.0f);
            return start + deltaMove;
        }
public void Update()
        {
            float deltaTime = UseUnscaledTime
                ? Time.unscaledDeltaTime
                : Time.deltaTime;

            bool interpOccuredThisFrame = false;

            if (AnimatingPosition)
            {
                Vector3 lerpTargetPosition = targetPosition;
                if (SmoothLerpToTarget)
                {
                    lerpTargetPosition = Vector3.Lerp(transform.position, lerpTargetPosition,
                     SmoothPositionLerpRatio);
                }

                Vector3 newPosition = NonLinearInterpolateTo(transform.position, lerpTargetPosition,
                deltaTime, PositionPerSecond);
                if ((targetPosition - newPosition).sqrMagnitude <= smallNumber)
                {
                    // Snap to final position
                    newPosition = targetPosition;
                    AnimatingPosition = false;
                }
                else
                {
                    interpOccuredThisFrame = true;
                }

                transform.position = newPosition;

                //calculate interpolatedVelocity and store position for next frame
                PositionVelocity = oldPosition - newPosition;
                oldPosition = newPosition;
```

```
}

// Determine how far we need to rotate
if (AnimatingRotation)
{
    Quaternion lerpTargetRotation = targetRotation;
    if (SmoothLerpToTarget)
    {
        lerpTargetRotation = Quaternion.Lerp(transform.rotation, lerpTargetRotation,
        SmoothRotationLerpRatio);
    }

    float angleDiff = Quaternion.Angle(transform.rotation, lerpTargetRotation);
    float speedScale = 1.0f + (Mathf.Pow(angleDiff, RotationSpeedScaler) / 180.0f);
    float ratio = Mathf.Clamp01((speedScale * RotationDegreesPerSecond * deltaTime) /
    angleDiff);

    if (angleDiff < Mathf.Epsilon)
    {
        AnimatingRotation = false;
        transform.rotation = targetRotation;
    }
    else
    {
        // Only lerp rotation here, as ratio is NaN if angleDiff is 0.0f
        transform.rotation = Quaternion.Slerp(transform.rotation, lerpTargetRotation, ratio);
        interpOccuredThisFrame = true;
    }
}

// Determine how far we need to rotate
if (AnimatingLocalRotation)
{
    Quaternion lerpTargetLocalRotation = targetLocalRotation;
    if (SmoothLerpToTarget)
    {
        lerpTargetLocalRotation = Quaternion.Lerp(transform.localRotation,
        lerpTargetLocalRotation, SmoothRotationLerpRatio);
    }

    float angleDiff = Quaternion.Angle(transform.localRotation, lerpTargetLocalRotation);
    float speedScale = 1.0f + (Mathf.Pow(angleDiff, RotationSpeedScaler) / 180.0f);
    float ratio = Mathf.Clamp01((speedScale * RotationDegreesPerSecond * deltaTime) /
    angleDiff);

    if (angleDiff < Mathf.Epsilon)
    {
        AnimatingLocalRotation = false;
        transform.localRotation = targetLocalRotation;
    }
    else
    {
        // Only lerp rotation here, as ratio is NaN if angleDiff is 0.0f
        transform.localRotation = Quaternion.Slerp(transform.localRotation,
        lerpTargetLocalRotation, ratio);
        interpOccuredThisFrame = true;
    }
}

if (AnimatingLocalScale)
{
    Vector3 lerpTargetLocalScale = targetLocalScale;
    if (SmoothLerpToTarget)
    {
        lerpTargetLocalScale = Vector3.Lerp(transform.localScale, lerpTargetLocalScale,
        SmoothScaleLerpRatio);
    }

    Vector3 newScale = NonLinearInterpolateTo(transform.localScale, lerpTargetLocalScale,
```

```
                deltaTime, ScalePerSecond);
                if ((targetLocalScale - newScale).sqrMagnitude <= smallNumber)
                {
                    // Snap to final scale
                    newScale = targetLocalScale;
                    AnimatingLocalScale = false;
                }
                else
                {
                    interpOccuredThisFrame = true;
                }

                transform.localScale = newScale;
            }

            // If all interpolations have completed, stop updating
            if (!interpOccuredThisFrame)
            {
                if (InterpolationDone != null)
                {
                    InterpolationDone();
                }
                enabled = false;
            }
        }

        /// <summary>
        /// Snaps to the final target and stops interpolating
        /// </summary>
public void SnapToTarget()
        {
            if (enabled)
            {
                transform.position = TargetPosition;
                transform.rotation = TargetRotation;
                transform.localRotation = TargetLocalRotation;
                transform.localScale = TargetLocalScale;

                AnimatingPosition = false;
                AnimatingLocalScale = false;
                AnimatingRotation = false;
                AnimatingLocalRotation = false;

                enabled = false;

        if (InterpolationDone != null)
                {
                    InterpolationDone();
                }
            }
        }

        /// <summary>
        /// Stops the interpolation regardless if it has reached the target
        /// </summary
public void StopInterpolating()
        {
            if (enabled)
            {
                Reset();

                if (InterpolationDone != null)
                {
                    InterpolationDone();
                }
            }
        }

        /// <summary>
```

```
        /// Stops the transform in place and terminates any animations.
        /// </summary>
public void Reset()
        {
            targetPosition = transform.position;
            targetRotation = transform.rotation;
            targetLocalRotation = transform.localRotation;
            targetLocalScale = transform.localScale;

            AnimatingPosition = false;
            AnimatingRotation = false;
            AnimatingLocalRotation = false;
            AnimatingLocalScale = false;

            enabled = false;
        }

        /// <summary>
        /// If animating position, specifies the target position as specified
        /// by SetTargetPosition. Otherwise returns the current position of
        /// the transform.
        /// </summary>
public Vector3 TargetPosition
        {
            get
            {
                if (AnimatingPosition)
                {
                    return targetPosition;
                }
                return transform.position;
            }
        }

        /// <summary>
        /// If animating rotation, specifies the target rotation as specified
        /// by SetTargetRotation. Otherwise returns the current rotation of
        /// the transform.
        /// </summary>
public Quaternion TargetRotation
        {
            get
            {
                if (AnimatingRotation)
                {
                    return targetRotation;
                }
                return transform.rotation;
            }
        }

        /// <summary>
        /// If animating local rotation, specifies the target local rotation as
        /// specified by SetTargetLocalRotation. Otherwise returns the current
        /// local rotation of the transform.
        /// </summary>
public Quaternion TargetLocalRotation
        {
            get
            {
                if (AnimatingLocalRotation)
                {
                    return targetLocalRotation;
                }
                return transform.localRotation;
            }
        }

        /// <summary>
```

```
        /// If animating local scale, specifies the target local scale as
        /// specified by SetTargetLocalScale. Otherwise returns the current
        /// local scale of the transform.
        /// </summary>
public Vector3 TargetLocalScale
        {
            get
            {
                if (AnimatingLocalScale)
                {
                    return targetLocalScale;
                }
                return transform.localScale;
            }
        }
    }
}
```

Appendix H: Robot Interaction Script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Box_Collide : MonoBehaviour {

    bool robothasJoint = false;
    bool fixture = false;
    public string dropoff = "Table Jig";
    public string SnapObject = "hub_knuckle_snapped";

    void OnCollisionEnter(Collision collision)
    {

        if (collision.gameObject.name == "Tool Flange" && robothasJoint == false && fixture== false)
        {
            //create
            gameObject.AddComponent<FixedJoint>();
            gameObject.GetComponent<FixedJoint>().connectedBody = collision.rigidbody;
            robothasJoint = true;
            Debug.Log("Hit");

        }
        if (collision.gameObject.name == dropoff && robothasJoint == true && fixture == false)
        {
            // Destroy(gameObject.GetComponent<FixedJoint>());
                robothasJoint = false;
                fixture = true;
                Debug.Log("break");
                gameObject.transform.localScale = new Vector3(0, 0, 0);
                float scale = 0.0254f;
                GameObject.Find(SnapObject).transform.localScale = new Vector3(scale, scale, scale);


        }
    }
}
```

Refences

[1] University of California San Diego et al. “A Roadmap for US Robotics from Internet to Robotics” 2016 ed., 2016

[2] A.B. Moniz and B.J. Krings, “Robots Working with Humans or Humans Working with Robots? Searching for Social Dimensions in New Human-Robot Interactions in Industry,” *Societies*, vol 6,no. 23, Aug. 2016

[3] B. Carney, "Industrial Robots And Automation - Humascend", Humascend, 2018. [Online]. Available: http://www.humascend.info/2017/06/13/industrial-robots-automation/.

[4] F. Hernoux et al., “Virtual Reality for Improving Safety and Collaborative Control of Industrial Robots” , April 2015

[5] T. Deyle, "Baxter Robot from Rethink Robotics Finally Unveiled! | Hizook", Hizook.com, 2018. [Online]. Available: http://www.hizook.com/blog/2012/09/18/baxter-robot-rethink-robotics-finally-unveiled.

[6] L. Dormehl, "8 virtual reality milestones that took it from sci-fi to your living room", Digital Trends, 2018. [Online]. Available: https://www.digitaltrends.com/cool-tech/history-of-virtual-reality/.

[7] C. Moore and V. Moore, "Microsoft’s HoloLens award recipients, Instagram client pulled from iOS store, and Microsoft’s Project Oxford SDKs—SD Times news digest: Nov. 12, 2015 - SD Times", SD Times, 2018. [Online]. Available: https://sdtimes.com/microsofts-hololens-award-recipients-instagram-client-pulled-from-ios-store-and-microsofts-project-oxford-sdks-sd-times-news-digest-nov-12-2015/.

[8] B. Christian, "HoloLens trial gives doctors 'X-ray vision' to allow them to peer inside patients during surgery", Wired.co.uk, 2018. [Online]. Available: http://www.wired.co.uk/article/industries-using-microsoft-hololens.

[9] "UR10 Collaborative industrial robotic arm - Payload up to 10 kg", Universal-robots.com, 2018. [Online]. Available: https://www.universal-robots.com/products/ur10-robot/.

[10] "What the #&@% is it?", Road & Track, 2018. [Online]. Available: https://www.roadandtrack.com/car-culture/car-accessories/reviews/a4258/boot-what-the-knuckle/.

[11] "Southeast Auto Service - Suspension & Front End Repair", Southeastautoservice.com, 2018. [Online]. Available: http://www.southeastautoservice.com/services/suspension.html.

[12] GrabCAD, 2018. [Online]. Available: https://grabcad.com/library.